НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА СИСТЕМНОГО ПРОГРАМУВАННЯ І
СПЕЦІАЛІЗОВАНИХ КОМП'ЮТЕРНИХ СИСТЕМ

«На правах рукопису»
УДК 000.428

«До захисту допущено»

Завідувач кафедри СПСКС

_____ В.П.Тарасенко_
(підпис) (ініціали, прізвище)

"___"_____2018р.

Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності 123 Комп'ютерна інженерія
Спеціалізовані комп'ютерні системи

на тему: ПРОГРАМНА БІБЛІОТЕКА ОБМІНУ ДАНИМИ МІЖ ПРОЦЕСАМИ

Виконав: студент ІІ курсу, групи КВ-73мп
(шифр групи)

Черниш Андрій Анатолійович_____ _____
(прізвище, ім'я, по батькові) (підпис)

Науковий керівник к.т.н., доц. Потапова К.Р._____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент_____ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2018 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих комп'ютерних систем

Рівень вищої освіти – другий (магістерський)
Спеціальність 123 Комп'ютерна інженерія
Спеціалізовані комп'ютерні системи

ЗАТВЕРДЖУЮ
Завідувач кафедри СПСКС
_____ Тарасенко В.П.
(підпис) (ініціали, прізвище)
"___" _____ 2018 р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Чернишу Андрію Анатолійовичу
(прізвище, ім'я, по батькові)

1. Тема дисертації «Програмна бібліотека обміну даними між процесами» науковий керівник дисертації доц.каф.СПСКС, к.т.н., Потапова К.Р. затверджені наказом по університету від 30 жовтня 2018 р. №4030-с.
2. Термін подання студентом дисертації 7 грудня 2018 р.
3. Об'єкт дослідження: обмін даними між процесами.
4. Предмет дослідження: методи та способи обміну даними між процесами засобами реактивного програмування.
5. Перелік завдань, які потрібно розробити: реалізувати обмін даними між процесами за допомогою сокетів, розробити програмну бібліотеку засобами реактивного програмування, розробити демонстраційний приклад використання програмної бібліотеки обміну даними між процесами.
6. Перелік ілюстративного матеріалу: презентація (кількість аркушів: 13)

7. Перелік публікацій: «Організація обміну даними між процесами засобами реактивного програмування», наукова конференція магістрантів та аспірантів «Прикладна математика та комп'ютинг» ПМК-2018 (Київ, 14-16 листопада 2018 р.); «Обмін даними між процесами засобами мови програмування Kotlin та парадигми реактивного програмування», V Міжнародна науково-технічна Internet-конференція «Сучасні методи, інформаційне, програмне та технічне забезпечення систем керування організаційно-технічними та технологічними комплексами» (Національний університет харчових технологій, листопад 2018 р.)

8. Дата видачі завдання 5 вересня 2017 .

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1.	Вивчення літератури за тематикою дисертації	10.09.2017	
2.	Підготовка матеріалів першого розділу магістерської дисертації	21.11.2017	
3.	Підготовка матеріалів другого розділу магістерської дисертації	14.02.2018	
4.	Підготовка матеріалів третього розділу магістерської дисертації	19.04.2018	
5.	Підготовка матеріалів четвертого розділу магістерської дисертації	05.05.2018	
6.	Підготовка матеріалів п'ятого розділу магістерської дисертації	30.06.2018	
7.	Розробка програмної бібліотеки	19.07.2018	
8.	Розробка моделі, що демонструє приклад використання програмної бібліотеки	07.09.2018	
9.	Оформлення документації магістерської дисертації	10.10.2018	
10.	Попередній розгляд магістерської дисертації на кафедрі	26.11.2018	

Студент _____ Черниш А.А.
(підпис) (прізвище, ініціали)

Науковий керівник дисертації _____ Потапова К.Р.
(підпис) (прізвище, ініціали)

РЕФЕРАТ

Актуальність теми. Реалізація обміну даними між комп’ютерними системами була завжди нетривіальною задачею для інженерів. При побудові коректної та оптимальної архітектури програмних продуктів важлива чітка читабельність коду програми та швидкість розробки. У сучасний період мають неабияку популярність програмні додатки для онлайн спілкування. Головна їхня суть – швидкий та надійний обмін повідомленнями між користувачами. Процес реалізації обміну даними комп’ютерних систем безупинно розвивається в плані швидкодії та швидкості розробки, проте він також повинен бути зрозумілим для всіх інженерів. В магістерській дисертації описується програмна бібліотека обміну даними між процесами, яка має на меті оптимізацію швидкодії, швидкості розробки, якості та чистоти програмного коду.

Об’єктом дослідження є обмін даними між процесами

Предметом дослідження є методи та способи обміну даними між процесами засобами реактивного програмування.

Мета роботи полягає у розробці програмної бібліотеки обміну даними між процесами засобами реактивного програмування.

Методи дослідження. Одним з найважливіших методів дослідження у роботі є аналіз та власне розробка, оскільки магістерська дисертація присвячена вивченню великих об'ємів накопичених знань у питаннях обміну даними між процесами, їх обробці та пошуку шляхів їх вдосконалення. Також, було використано метод абстрагування, що дозволяє виділяти для використання лише певний набір властивостей об'єкта, ігноруючи інші, не важливі для користувача зв'язки та відношення.

Наукова новизна роботи полягає в тому, що розроблена програмна бібліотека має вищий рівень абстракції програмного коду обміну даними

між процесами, а також містить ширшу функціональність, порівняно з аналогами, завдяки використанню парадигми реактивного програмування.

Практична цінність отриманих в роботі результатів полягає в тому, що запропонована програмна бібліотека реалізована для використання реальними програмними продуктами, що можуть бути впроваджені для реальних комп'ютерних систем.

Апробація роботи. Результати роботи пройшли апробацію або знаходяться на стадії публікації на конференціях:

– XI конференція молодих вчених «Прикладна математика та комп'ютинг» ПМК-2018-2;

– V міжнародна науково-технічна Internet-конференція «Сучасні методи, інформаційне, програмне та технічне забезпечення систем керування організаційно-технічними та технологічними комплексами», яка проводилась 22 листопада 2018 р. у Національному університеті харчових технологій.

Структура та обсяг роботи. Магістерська дисертація складається з вступу, п'ятьох розділів, висновків та додатків.

У вступі надано загальну характеристику роботи, виконано оцінку сучасного стану проблеми, обґрунтовано актуальність напрямку досліджень, сформульовано мету і задачі дослідження та розробки.

У першому розділі описується організація процес обміну даними між процесами; розглянуто основні протоколи на різних рівнях моделі OSI; описана теоретична частина таких видів архітектур як: клієнт-серверна архітектура, архітектура REST, протокол HTTP та принципи роботи сокетів;

У другому розділі обрано парадигму програмування, згідно з якою розроблена архітектура програмної бібліотеки; описана теоретична частина обраної парадигми реактивного програмування; детальна аргументація

використаної мови програмування Kotlin, її актуальність, універсальність та переваги перед іншими мовами.

У третьому розділі запропоновано структуру та алгоритми реалізації програмної бібліотеки; описана Domain-Specific Language мовою Kotlin для ініціалізації програмної бібліотеки; наведені поля та методи налаштування конфігурації сокетів для коректної передачі даних у різних варіаціях використання.

У четвертому розділі описується методика використання програмної бібліотеки, створеної у даній магістерській дисертації; перечислені основні методи та поля для створення екземпляру сокета, відправки даних, реєстрації власно визначених користувачем подій, а також закриття та видалення екземпляру сокета.

У п’ятому розділі наведено приклад використання програмної бібліотеки обміну даними між процесами; описана програмно-апаратна модель реального використання програмного продукту на практиці.

У висновках представлено отримані результати проведеної роботи.

У додатках наведено фрагменти реалізації програмної бібліотеки, фрагменти прикладу використання програмної бібліотеки, копії публікацій, довідка про впровадження та копії графічних матеріалів.

Магістерська дисертація виконана на 87 аркушах, містить 5 додатків та посилання на список використаних літературних джерел з 18 найменувань. У роботі наведено 21 рисунок та 2 таблиці.

Ключові слова: обмін даними між процесами, реактивне програмування, Kotlin, сокет.

РЕФЕРАТ

Актуальность темы. Реализация обмена данными между компьютерными системами была всегда нетривиальной задачей для инженеров. При построении корректной и оптимальной архитектуры программных продуктов важна четкая читабельность кода программы и скорость разработки. В современный период имеют большую популярность программные приложения для онлайн общения. Главная их суть – быстрый и надежный обмен сообщениями между пользователями. Процесс реализации обмена данными компьютерных систем непрерывно развивается в плане быстродействия и скорости разработки, однако, он также должен быть понятным для всех инженеров. В магистерской диссертации описывается программная библиотека обмена данными между процессами, которая имеет целью оптимизации быстродействия, скорости разработки, качества и чистоты кода.

Объектом исследования является обмен данными между процессами.

Предметом исследования являются методы и способы обмена данными между процессами средствами реактивного программирования.

Цель работы заключается в разработке программной библиотеки обмена данными между процессами средствами реактивного программирования.

Методы исследования. Одним из важнейших методов исследования в работе является анализ и собственно разработка, так как магистерская диссертация посвящена изучению больших объемов накопленных знаний в вопросах обмена данными между процессами, их обработке и поиска путей их совершенствования. Также, был использован метод абстрагирования, позволяющего выделять для использования только определенный набор свойств объекта, игнорируя другие, не важны для пользователя связи и отношения.

Научная новизна работы заключается в том, что разработана программная библиотека, имеет высокий уровень абстракции программного кода обмена данными между процессами, а также содержит широкую функциональность по сравнению с аналогами, благодаря использованию парадигмы реактивного программирования.

Практическая ценность полученных в работе результатов заключается в том, что предложенная программная библиотека реализована для использования реальными программными продуктами, которые могут быть внедрены для реальных компьютерных систем.

Апробация работы. Результаты работы прошли апробацию или находятся на стадии публикации на конференциях:

– XI конференция молодых ученых «Прикладная математика и компьютинг» ПМК-2018-2;

– V международная научно-техническая Internet-конференция «Современные методы, информационное, программное и техническое обеспечение систем управления организационно-техническими и технологическими комплексами», которая проводилась 22 ноября 2018 в Национальном университете пищевых технологий.

Структура и объем работы. Магистерская диссертация состоит из введения, пяти глав, заключения и приложений.

Во введении дана общая характеристика работы, выполнена оценка современного состояния проблемы, обоснована актуальность направления исследований, сформулированы цели и задачи исследования и разработки.

В первом разделе описывается организация процесс обмена данными между процессами; рассмотрены основные протоколы на разных уровнях модели OSI; описана теоретическая часть таких видов архитектур как: клиент-серверная архитектура, архитектура REST, протокол HTTP и принципы работы сокетов;

Во втором разделе избрана парадигма программирования, согласно которой разработана архитектура программной библиотеки; описана теоретическая часть выбранной парадигмы реактивного программирования; подробная аргументация использованного языка программирования Kotlin, его актуальность, универсальность и преимущества перед другими языками.

В третьем разделе предложена структура и алгоритмы реализации программной библиотеки; описана Domain-Specific Language языке Kotlin для инициализации программной библиотеки; приведены поля и методы настройки конфигурации сокетов для корректной передачи данных в различных вариациях использования.

В четвертом разделе описывается методика использования программной библиотеки, созданной в данной магистерской диссертации; перечислены основные методы и поля для создания экземпляра сокета, отправки данных, регистрации собственно определенных пользователем событий, а также закрытие и удаление экземпляра сокета.

В пятом разделе приведен пример использования программной библиотеки обмена данными между процессами; описана программно-аппаратная модель реального использования программного продукта на практике.

В выводах представлены полученные результаты проведенной работы.

В приложениях приведены фрагменты реализации программной библиотеки, фрагменты примера использования программной библиотеки, копии публикаций, справка о внедрении и копии графических материалов.

Магистерская диссертация выполнена на 87 страницах, содержит 5 приложений и ссылки на список использованных литературных источников из 18 наименований. В работе приведены 21 рисунок и 2 таблицы.

Ключевые слова: обмен данными между процессами, реактивное программирование, Kotlin, сокет.

ABSTRACT

Actuality of theme. The implementation of data exchange between computer systems has always been a non-trivial task for engineers. Constructing the correct and optimal software architecture, there are very important readability of the program code and the performance of development. Nowadays, software applications for online communication are very popular. Their main essence is fast and reliable messaging between users. The process of implementing computer data exchanging is constantly evolving in terms of performance and development speed, but it should also be understandable for all engineers. The master's dissertation describes a software library for exchanging data between processes, which aims to optimize performance, speed of development, quality and cleanliness of the code.

The object of the study is data exchange between processes.

The subject of the study is methods of data exchange between processes by means of reactive programming.

The purpose of the work: to develop a software library for data exchange between processes by means of reactive programming.

Scientific novelty of the work is that the software library has been developed in master’s dissertation has a high level of abstraction of the code of data exchange between processes. In addition, it contains a wide functionality in comparison with analogues, because of reactive programming paradigm.

The practical value of the results obtained in the work is that the proposed software library is developed for use by real software products that can be deployed to real computer systems.

Test work. The proposed software library was presented and discussed at the conferences:

– Scientific conference of undergraduates and postgraduates "Applied Mathematics and Computing", PMK-2018 (Kyiv, November 14-16, 2018)

– V International Scientific and Technical Internet Conference "Modern Methods, Information, software and technical support of control systems for organizational, technical and technological complexes", held on November 22, 2018 at the National University of Food Technologies.

Structure and scope of work. The master's dissertation consists of an introduction, five chapters, conclusions and five appendences.

The introduction gives a general characteristic of work, an assessment of the current state of the problem, the relevance of the direction of research, the goals and objectives of research and development are formulated.

The first chapter describes the organization of the process of data exchange between processes; consider the main protocols at different levels of the OSI model; describes the theoretical part of such architectures as client-server architecture, REST architecture, HTTP protocol and socket operation principles;

The second chapter selects a paradigm of programming, according to which the architecture of the software library was developed; describes the theoretical part of the chosen reactive programming paradigm; detailed argumentation of the used Kotlin programming language, its relevance, versatility and advantages over other programming languages.

The third chapter proposes structure and algorithms of implementation of the software library; Kotlin Domain-Specific Language is described for initializing the software library; the fields and methods for configuring sockets for correct data exchange in different usage variations are given.

The fourth chapter describes the methodology of using the software library, which was created in this master's dissertation; describes the main methods and fields for creating a socket instance, sending data, registering user-defined events, closing and deleting an instance of a socket.

The fifth chapter describes an example of using a software library for data exchange between processes; describes the hardware and software model of real use of the software product in practice.

The conclusions presents the obtained results of the conducted work.

The appendices includes the parts of the implementation of the software library, fragments of the software library use example, a copy of the publications, the software library introduction certificate and a copies of the graphic materials.

The master's dissertation is executed on 87 pages, contains 5 appendices and links to the list of used literary sources from 18 titles. In the work 21 figures and 2 tables are given.

Keywords: data exchange between processes, reactive programming, Kotlin, socket.

ЗМІСТ

ДОДАТКИ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ, ПОЗНАЧЕНЬ, ТЕРМІНІВ

Бібліотека – збірка об'єктів чи підпрограм для вирішення близьких за тематикою задач.

ЕОМ – електро обчислювальна машина – загальна назва для обчислювальних машин, що є електронними (починаючи з перших лампових машин, включаючи напівпровідникові, тощо) на відміну від електромеханічних та механічних обчислювальних машин.

ОС – операційна система – це базовий комплекс програм, що виконує управління апаратною складовою комп'ютера або віртуальної машини.

ПЗ – програмне забезпечення.

Паттерн проектування – ефектний спосіб вирішення задач проектування при розробці програмного забезпечення.

СУБД – система управління базами даних – комплекс програмного забезпечення, що надає можливості створення, збереження, оновлення та пошуку інформації в базах даних з контролем доступу до даних.

Сокет – назва програмного інтерфейсу для забезпечення обміну даними між процесами.

Фреймворк – інфраструктура програмних рішень, що полегшує розробку складних систем. Спрощено дану інфраструктуру можна вважати своєрідною комплексною бібліотекою.

Android – операційна система для мобільних пристроїв.

Android Studio – середовище розробки додатків на мобільні пристрої з операційою системою Android.

API – Application Programming Interface – прикладний програмний інтерфейс. Набір визначень підпрограм, протоколів взаємодії та засобів для створення програмного забезпечення.

GPIO – General Purpose Input Output – інтерфейс введення/виведення загального призначення. Інтерфейс для зв'язку

між компонентами комп'ютерної системи, наприклад, мікропроцесором і різними периферійними пристроями.

HTTP – Hyper Text Transfer Protocol – протокол передачі гіпер-текстових документів, що використовується в комп’ютерних мережах.

IDE – Integrated Development Environment – інтегроване середовище розробки. Комплексне програмне рішення для розробки програмного забезпечення.

Java – мова програмування високого рівня.

Pi4J – бібліотека з відкритим вихідним кодом для розробки програмних додатків для одноплатних комп’ютерів Raspberry Pi.

JSON – JavaScript Object Notation – запис об'єктів JavaScript. Текстовий формат обміну даними між комп'ютерами

Kotlin – мова програмування високого рівня.

Raspbian OS – Raspbian Operating System – операційна система, що підтримується одноплатними комп’ютерами Raspberry Pi.

REST – Representational State Transfer – передача репрезентативного стану. Підхід до архітектури мережевих протоколів, які забезпечують доступ до інформаційних ресурсів.

RPi3 – одноплатний комп’ютер Raspberry Pi 3.

SDK – Software Development Kit – набір із засобів розробки, утиліт і документації, який дозволяє програмісту створювати прикладні програми за визначеною технологією або для певної платформи (програмної або програмно-апаратної).

URI – Uniform Resource Identifier – уніфікований ідентифікатор ресурсів. Компактний рядок літер, який однозначно ідентифікує окремий абстрактний чи фізичний ресурс.

XML – Extensible Markup Language – розширювана мова розмітки. Стандарт побудови мов розмітки ієрархічно структурованих даних для обміну між різними застосунками, зокрема, через Інтернет.

ВСТУП

Реалізація обміну даними між комп'ютерними системами була завжди нетривіальною задачею для інженерів. При побудові коректної та оптимальної архітектури програмних продуктів важлива чітка читабельність коду програми та швидкість розробки. Будь-який процес повинен обмінюватись інформацією з іншими процесами як на одній, так і на різних ЕОМ, пов'язаних між собою мережею. Процес реалізації обміну даними комп'ютерних систем необхідно оптимізувати в плані швидкодії та швидкості розробки, проте він також повинен бути зрозумілим для всіх інженерів.

Існуючі реалізації обміну даними на транспортному рівні являються складними в реалізації, а на прикладному рівні є втрати швидкодії та деяких функцій. За допомогою програмного інтерфейсу обміну даними, розробленого в даній магістерській дисертації можна буде з легкістю використовувати звичайні сокети на вищому рівні абстракції. Таким чином реалізація передачі інформації займатиме набагато менше часу. Передача буде універсальною між будь-яким процесом, що виконується на віртуальній машині Java. Швидкість передачі інформації буде вищою ніж через інтерфейси прикладного рівня, такі як HTTP.

За останні 5 років парадигма реактивного програмування набула неабиякої популярності не лише в розробці мобільних додатків, а й у веб-розробці [1]. Вона є чудовою реалізацією автоматизації в розсиланні змін через потік даних. Наприклад, у архітектурі Модель-Представлення-Контролер, реактивне програмування дозволяє змінам у «моделі» автоматично відображатися у «представленні», і навпаки. Реактивне програмування має принципові подібності до шаблону проектування «спостерігач», що зазвичай використовують в обєктно-орієнтованому програмуванні (ООП).

Мова програмування Java й досі посідає перші місця на ринку комерційного виробництва програмного забезпечення. Дана мова програмування так широко використовується через надійність та стійкість

готових продуктів. Вона використовується не лише в серверній, але й в мобільній розробці, а програмний код, скомпільований на одній операційній системі – буде працювати на іншій, завдяки віртуальній машині Java (Java Virtual Machine – JVM) [2].

За всі роки використання Java люди помітили деякі синтаксичні недоліки а також помітили, що таку мову можна значно покращити. Ініціаторами змін стала компанія JetBrains, яка випустила нову мову програмування – Kotlin. Це мова, яка має багато нових покращень та особливостей, які не має Java, проте вона також працює поверх JVM і при компіляції компілюється в байткод. Тобто, ми можемо запускати додаток на Kotlin скрізь, де встановлена JVM, як і у випадку з Java. Також, можна компілювати код в JavaScript і запускати в браузері. Окрім того, можна компілювати Kotlin код в нативні бінарні файли, які будуть працювати без всяких віртуальних машин. Таким чином, круг платформ, для яких можна створювати програмні продукти на Kotlin надзвичайно широкий: Windows, Linux, Mac OS, iOS, Android [3].

Kotlin стрімко набуває популярності, останні статистики показують, що ця мова програмування входить в топ 10 набуваючих популярності. А з 17 травня 2017 року входить в список офіційно підтримуваних мов для розробки додатків для платформи Android [3]. В Україні все більше компаній розпочинають свої комерційні проекти з використанням даної мови програмування. Більш того, програмні застосунки для онлайн спілкування (Telegram, Messenger, Viber) є дуже популярними, а для їхньої коректної та швидкої роботи просто необхідна робота з сокетами.

Програмна бібліотека, розроблена в магістерській дисертації може використовуватись скрізь, де буде необхідно швидко реалізувати деяку частину роботи з сокетами. Вона адаптована для мови Kotlin і підтримує всі її особливості, проте її можна буде з легкістю використовувати засобами Java. Саме тому програмний продукт, розроблений в даній магістерській дисертації вважається актуальним.

1. ОРГАНІЗАЦІЯ ОБМІНУ ДАНИМИ МІЖ ПРОЦЕСАМИ

Будь-який процес повинен обмінюватись інформацією з іншими процесами як на одній, так і на різних ЕОМ, пов’язаних між собою мережею. Такий процес може виконуватись на різних операційних системах. Однією з основних задач магістерської дисертації є організація обміну між процесами так, щоб це можна було виконувати на переважній більшості операційних систем, оскільки це забезпечить універсальність використання продукту. Для цього було проаналізовано такі підходи міжпроцесної взаємодії:

- файл;
- база даних;
- сокети;
- обмін повідомленнями;
- HTTP (REST архітектура);
- Mailslot;

Файл – іменована область даних на носії інформації. В залежності від файлової системи, файл може мати різний набір властивостей. Він може мати ім’я, розширення, дату створення, час останнього доступу, права доступу, тощо. Над файлом можна виконувати такі операції: запис, читання, відкриття, закриття, перенесення вказівника. За допомогою файлу можна записати дані, що використовує один процес, та зчитати їх за допомогою іншого процесу [4]. В такому випадку можуть виникати проблеми з одночасним доступом до спільних даних, а також можуть виникати колізії. Такі проблеми можуть вирішитись за допомогою бази даних (БД).

База даних – впорядкований набір даних, що організовані відповідно до їх характеристик та взаємозв’язків між їх елементами. В загальному випадку база даних містить схеми, таблиці, збережені процедури, подання та інші об’єкти. В такому випадку, база даних містить опис та засоби обробки даних, що в ній зберігаються [5]. В сучасній комерційній розробці для забезпечення роботи з базами даних використовують системи керування базами даних (СУБД). Це

такі системи, що забезпечують доступ, контроль, створення, визначення та використання баз даних. Сучасні СУБД забезпечують такі функції: оголошення, отримання, модифікація та адміністрування даних. В кінцевому результаті база даних є файлом, на відміну від якого БД відрізняється оголошенням та використанням даних [5]. А отже в БД є функції, що забезпечують обробку колізій, які можуть виникнути при одночасному доступі до даних з різних процесів. Бази даних є дуже зручними та функціональними, проте вони зберігаються на одній ЕОМ, а отже обмін даними між процесами може відбуватись лише на одному комп'ютері. Саме тому БД використовують при розробці клієнт-серверної архітектури.

Mailslot (мейлслот) – механізм міжпроцесної взаємодії, що забезпечує однонаправлену передачу інформації, а також дозволяє здійснювати трансляцію повідомлень по мережі. Даний механізм являється клієнт-серверним інтерфейсом, тому повинен мати процес (сервер), який створює мейлслот та може читати з нього інформацію. Клієнтом мейлслоту може бути будь-який процес, що знає його ім'я. Сервером і клієнтом може бути один і той же процес [4]. В даній магістерській дисертації цей метод не розглядається, оскільки він забезпечує лише однонаправлену передачу інформації, що не є достатньо функціональним при розробці великих систем. Також, основною проблемою такого методу є те, що він підтримується переважно на операційній системі MS Windows, що не є універсальним підходом.

Сокети та HTTP (REST архітектура) були проаналізовані, як основні компоненти, за допомогою яких можливо найкраще організувати обмін даними між процесами. В таких випадках обмін відбуватиметься на будь-якій операційній системі, та підтримуватиметься клієнт-серверна архітектура, яка включає в себе шар баз даних.

Отже, як зазначалось вище, обмін даними між процесами повинен відбуватись як на одній ЕОМ, так і на декількох, що пов'язані між собою

мережею. Тому, в будь-якому випадку необхідна реалізація клієнт-серверної архітектури.

1.1. Клієнт-серверна архітектура

Клієнт і сервер за своєю природою є програмами. На перший погляд ці два поняття не є цілком зрозумілими, спробуємо навести приклади.

У повсякденному житті ми зустрічаємо клієнтські програми дуже часто, а саме у смартфонах та веб-браузерах. Мобільні додатки з широкою функціональністю не можуть містити лише статичні сторінки, та дані, які збережені локально у СУБД. Їм необхідно отримувати інші дані з мережі, оскільки вони є спільними для інших користувачів. Веб-браузер відкриває веб-сторінки, які теж шлють запити на сервер для отримання даних. Також, прикладом є переважна більшість СУБД, які мають клієнтські додатки, що забезпечують доступ до даних через графічний інтерфейс, який дозволяє керувати даними просто за допомогою периферійних пристроїв. У загальному випадку клієнт – це програмний додаток, що відправляє запити на сервер з метою обміну даними [6].

Щодо серверних додатків, то це спеціалізовані програми, що приймають запити клієнтів, обробляють їх, виконують деякі операції над ними, та мають змогу повертати дані до кожного з клієнтів. Така модель взаємодії між клієнтом та сервером призначена для того, щоб розподілити навантаження та функціональність між ними. Тому, клієнтський і серверний додаток можуть бути як на різних комп’ютерах, так і на одному і при цьому успішно взаємодіяти один з одним.

Прикладами серверних додатків є:

- сервер баз даних будь-якої СУБД;
- готові збірки для веб-розробника, такі як Denwer або локальний сервер AMPPS;
- HTTP сервер, наприклад: Tomcat, Apache, тощо [6].

Серверний додаток, на відміну від клієнтського, виконує ряд функцій, що дозволяють багатьом клієнтам відправляти та отримувати дані від лише одного сервера. Наприклад, сервер повинен приймати HTTP запити від клієнтів та аналізувати їх, перевіряючи отримані HTTP методи і поля заголовків, потім виконувати дії, зазначені в запиті і повертати клієнтам результати своєї роботи за допомогою спеціального HTTP повідомлення, яке називається відповідь.

Обмін даними між клієнтом та сервером відбувається за допомогою різних мережевих протоколів, навіть якщо обидва додатки встановлені на одній ЕОМ, наприклад, по протоколу HTTP, IP, FTP, або WebSocket. Наприклад, використовуючи HTTP протокол клієнт відправляє спеціальне HTTP повідомлення, в якому визначено які дані та в якому форматі він хоче отримати. Сервер, отримавши таке повідомлення, відсилає клієнту у відповідь схоже по структурі повідомлення, в якому міститься необхідна інформація [6].

Апаратне забезпечення клієнта та сервера може відрізнятись. Оскільки сервер обробляє більше інформації, то йому необхідно суттєво більше ресурсів. Проте, при правильному проектування архітектури взаємодії між додатками таке навантаження на сервер можна суттєво зменшити, наприклад, використовуючи сокети для обміну даними між ними. Варто зауважити, що серверу та клієнту не обов’язково необхідно бути на різних ЕОМ, вони можуть бути встановлені на одному й тому ж комп’ютері. На рис. 1.1 зображено спрощену схему взаємодії клієнта та сервера.

На рисунку можна помітити особливість, що до одного сервера може звертатись відразу декілька клієнтів, які можуть одночасно відправляти запити до нього. Це демонструє те, що сервер виконує задачі відразу в декількох потоках, що необхідно враховувати при проектуванні серверного додатку, а саме при розробці доступу до спільних даних.

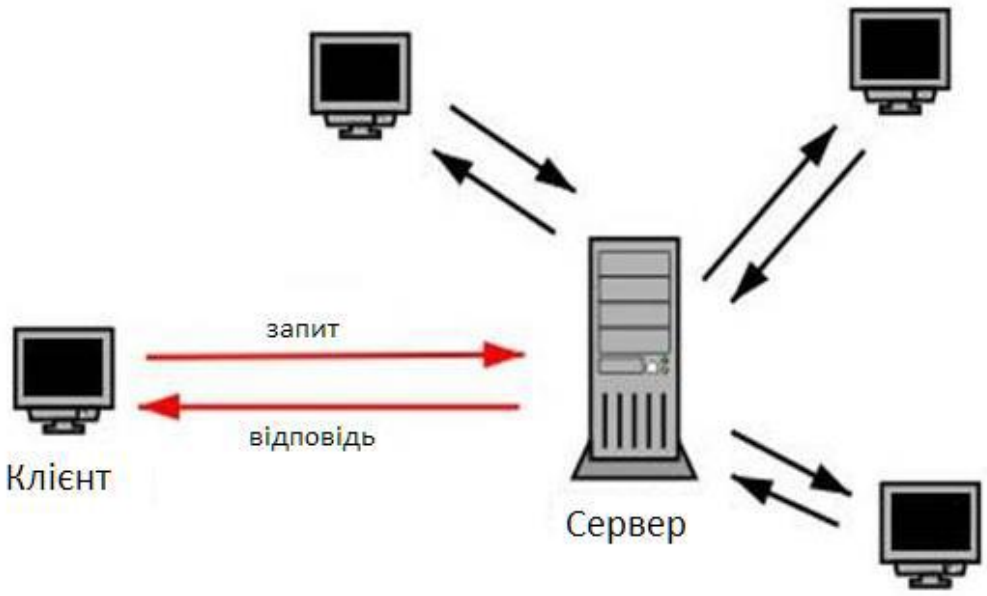

Рисунок 1.1 – Спрощена схема взаємодії між клієнтом та сервером

Багато мережевих протоколів побудовані на архітектурі клієнт-сервер, тому в їх основі зазвичай лежать однакові або схожі принципи взаємодії, а різниця лише в деталях, які обумовлені особливостями і специфікою області, згідно якої розроблявся той чи інший мережевий протокол.

Варто відзначити, що існує два види архітектури взаємодії клієнт-сервер: дворівнева та багаторівнева моделі.

Принципи роботи дворівневої моделі полягають в тому, що обробка запиту клієнта відбувається лише на одній ЕОМ, без використання сторонніх ресурсів. В такому випадку, архітектура вимагає сильне апаратне забезпечення, що зможе підтримувати надійність і швидкість обробки запитів. Таку модель зображено на рис. 1.2.

Принцип роботи багаторівневої моделі полягають в тому, що запити клієнтів можуть оброблятись відразу декількома серверами. Це робиться для того, щоб розподілити навантаження з одного сервера на декілька. В такому випадку зовсім не обов’язково мати потужне апаратне забезпечення для підтримки надійності та швидкості виконання запитів. На рис. 1.3 можна побачити схематичне зображення багаторівневої архітектури клієнт-сервер.

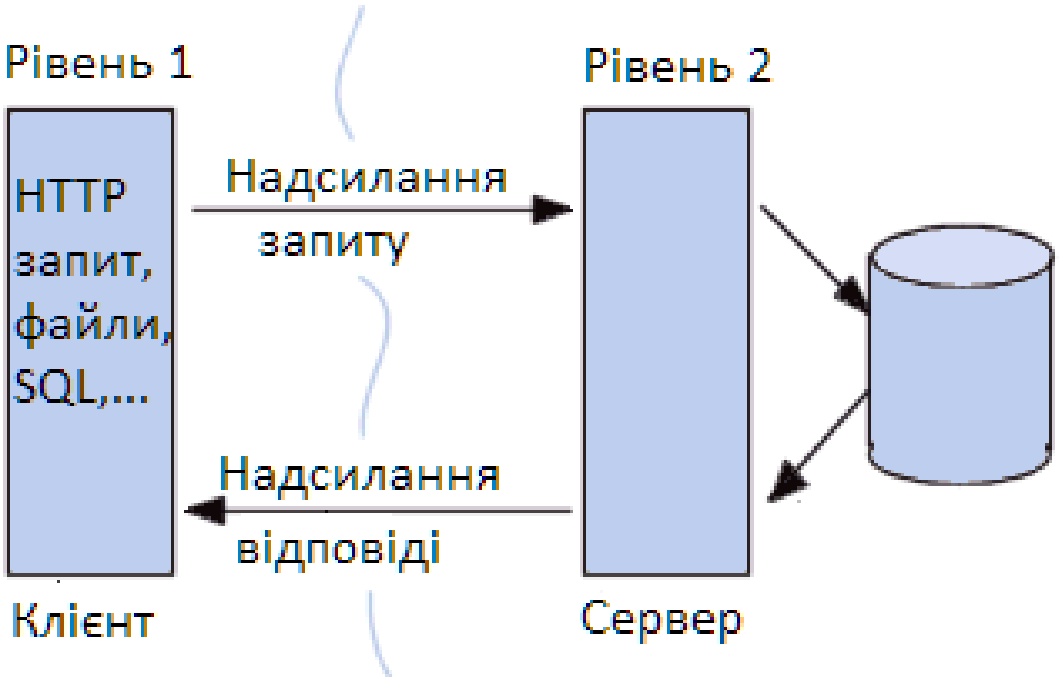

Рисунок 1.2 – Дворівнева модель взаємодії клієнт-сервер.

На рис. 1.2. видно, що є користувач (клієнт), що може запитувати або надсилати дані, і є сервер, що аналізує та обробляє ці дані згідно визначених правил, а також зберігає їх в базі даних.

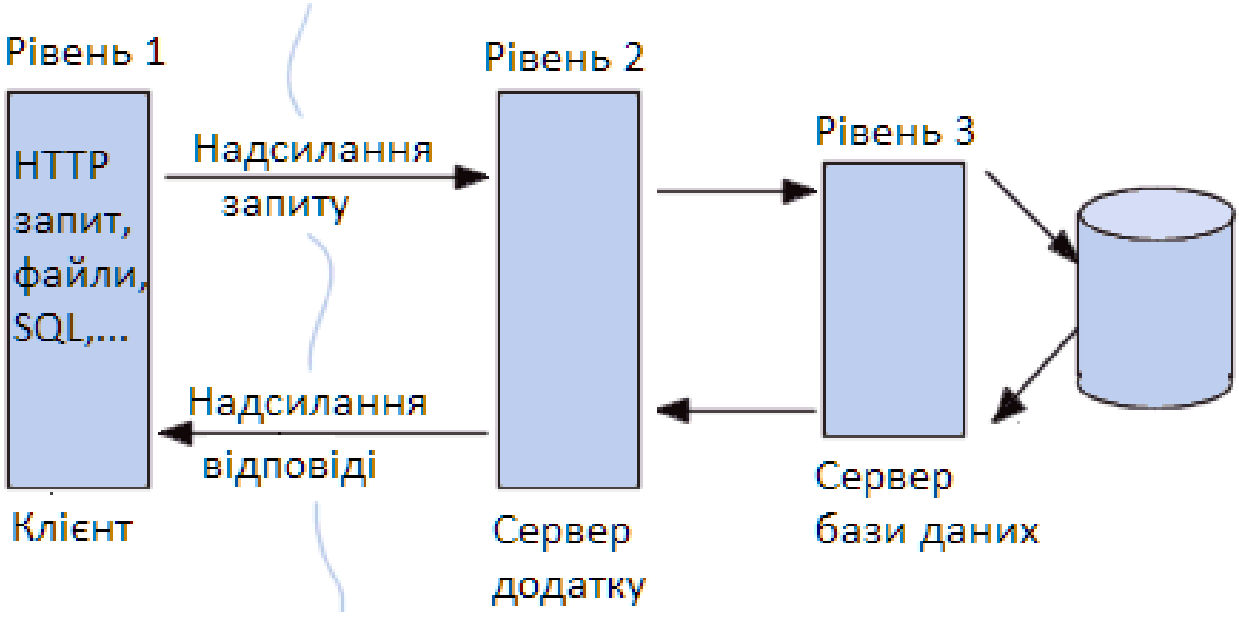

Рисунок 1.3 – Багаторівнева модель взаємодії клієнт-сервер.

Проте, варто зазначити, що передача даних між декількома серверами може займати визначений час, тому це необхідно враховувати при

проектуванні програмних додатків згідно технічного завдання. При такому підході сервер баз даних краще всього зосереджувати на окремій ЕОМ.

Перевагою моделі клієнт-сервер є те, що бізнес логіка, а також програмний код клієнтського та серверного додатків розділений. Це забезпечує зниження вимог до апаратного забезпечення ЕОМ, так як більша частина складних операцій відбувається на серверній частині. Варто відзначити, що архітектура клієнт-сервер досить гнучка, що забезпечує легке розширення функціональності, меншу зв'язність коду, та забезпечення захищеності локальної мережі.

До недоліків архітектури клієнт-сервер можна віднести те, що вартість серверного апаратного забезпечення значно вища ніж клієнтського. Також, проектування двох окремих додатків займатиме більше часу та матеріальних витрат зі сторони замовника. Варто зауважити, що при відмові серверної частини, клієнтські додатки не зможуть повністю функціонувати, для цього їм необхідно забезпечити правильне кешування даних, які зазвичай отримуються з сервера.

Суть концепції клієнт-сервер полягає не в конкретній реалізації. Вона визначає загальні принципи обміну даними між процесами, а самі деталі взаємодії визначають конкретні мережеві протоколи, що використовуються в певній реалізації. Дана концепція полягає в тому, що необхідно розділяти ЕОМ в мережі на клієнтські, яким завжди щось треба, і на серверні, які дають те, на що запитують клієнти [6].

1.2. Сокети

Розібравшись з клієнт-серверною архітектурою перейдемо до однієї з її реалізацій, а саме до сокетів. Перш за все необхідно визначити що це взагалі таке.

Сокет – кінцевий пункт двостороннього з'єднання між процесами, що комунікують в одній мережі. В комп'ютерних мережах таке двостороннє з'єднання прийнято вважати повним дуплексним. Завдяки сокетам можна не

просто обмінюватись звичайними повідомленнями, а й відправляти графічні та медіа файли, представивши їх у вигляді бінарних файлів.

Сокети є розширенням протоколу HTTP, який є синхронним і реалізує модель «запит - відповідь». При реалізації сокетів пропадає чітке розмежування клієнту та серверу, оскільки вони стають рівноправними учасниками обміну даними. Кожен працює сам по собі і відправляє дані лише тоді, коли це необхідно. При тому кожен може приймати дані в будь-який момент. Так реалізується асинхронність обміну даними між процесами.

Хоч в сокетах і немає чіткого розмежування між клієнтом та сервером, проте в програмній реалізації є відмінності. Тому, програмно необхідно створювати різні кінцеві точки в залежності від задачі, що стоїть перед розробником. Наприклад: якщо необхідно реалізувати багатокористувацький чат, тоді сервер, матиме список клієнтських сокетів, яким необхідно відправляти повідомлення, а клієнти відправлятимуть не один одному повідомлення, а на сервер, вказуючи отримувача.

Для взаємодії між процесами за допомогою стеку протоколів TCP/IP використовуються адреси та порти. Адресою є 32-бітна структура для протоколу IPv4, або ж 128-бітна для протоколу IPv6. Номер порту – ціле число в діапазоні від 0 до 65535 (для протоколу TCP). Наприклад, при зверненні до сервера на HTTP-порт сокет буде виглядати так: 194.135.12.15:80

Алгоритм обміну даними між процесами за допомогою сокетів можна описати за допомогою таких пунктів:

1. Створення екземпляру сокета.
2. Задання конфігурації, а саме: кінцевої IP адреси серверу (або домену), порту, забезпечення можливості кешування з’єднання, встановлення транспортного протоколу, забезпечення можливості повторного з’єднання, встановлення кількості спроб повторного з’єднання, тощо.
3. Реєстрація подій, пов’язаних зі станом сокета, а саме: створення, розриву, помилки створення, повтору створення з’єднання, тощо.
4. Реєстрація подій, визначених розробником.

5. Відправлення даних через сокет.
6. Закриття з’єднання та видалення сокету.

Передача даних за допомогою сокетів є швидкою, оскільки встановлення з’єднання відбувається лише раз, на відміну від протоколу HTTP. Високу швидкість та ефективність передачі забезпечує малий розмір даних, що передаються. Вони можуть навіть вміститись в один TCP пакет, хоча все залежить від бізнес логіки програмного додатку.

Сокети можуть використовуватись в розробці:

- додатків, що працюють в режимі реального часу;
- чат-додатків;
- IoT-додатків;
- багатокористувацьких ігор.

При створенні звичайних статичних веб сторінок сокети не є рішенням проблеми, оскільки в таких випадках немає необхідності асинхронного обміну даними. Проте, при проектування складних систем, у яких необхідна швидка, надійна, асинхронна передача даних - сокети стануть у нагоді.

1.3. Протокол HTTP

Абревіатура HTTP розшифровується як HyperText Transfer Protocol, тобто «протокол передачі гіпертекстових документів». З самого початку цей протокол назначався для передачі саме гіпертекстових документів, тобто таких, що можуть містити посилання, що позволяють перейти до інших документів [7].

Згідно специфікації моделі OSI, HTTP є протоколом верхнього, прикладного, 7-го рівня. Він передбачає реалізації клієнт-серверної архітектури передачі даних. Клієнтський додаток формує запит, відправляє його на сервер, після чого серверний додаток аналізує інформацію в заголовках, тілі та назві методу, отримує дані, обробляє їх згідно специфікацій бізнес логіки та повертає результат клієнту. Після отримання даних клієнтський додаток може відправляти інші запити до сервера.

API багатьох програмних продуктів передбачають використання HTTP протоколу для передачі даних, при чому самі дані можуть бути в будь-якому форматі, наприклад, JSON або XML. Також, HTTP може використовуватись іншими протоколами прикладного рівня, такими як SOAP, WebDAV, XML-RPC. В такому випадку кажуть, що він використовується як «транспорт».

Задача, яка традиційно вирішується за допомогою протоколу HTTP – це передача інформації між клієнтським додатком (наприклад: смартфон, веб-браузер) та серверним додатком. На даний момент всесвітня павутина працює завдяки використанню саме цього протоколу.

Як правило, передача по протоколу HTTP відбувається через TCP/IP з'єднання, при чому серверний додаток зазвичай використовує TCP-порт 80, хоча можна використовувати будь-який інший.

URI (Uniform Resource Identifier – унікальний ідентифікатор ресурсу) – шлях до конкретного ресурсу над яким необхідно виконати операцію. Деякі запити можуть не відноситись ні до якого ресурсу, тому в такому випадку замість URI пишеться зірочка (символ «*») [7].

Основним об'єктом маніпуляції HTTP є ресурс, на який вказує URI в запиті клієнту. Протокол HTTP дозволяє вказувати спосіб представлення одного і того ж ресурсу за різними параметрами, наприклад, мові. Завдяки цьому клієнт та сервер можуть обмінюватись двійковими даними, хоча даний протокол являється текстовим.

Кожен HTTP запит складається з трьох частин, які передаються в такому порядку:

– стартова строка – визначає тип повідомлення;

– заголовки – строки, що містять пару параметр-значення, які розділені між собою двокрапкою. Заголовки повинні відділятись від тіла хоча б однією пустою строкою. В заголовках містяться параметри даних, інші відомості;

– тіло повідомлення – обов'язково повинні відділятись пустою строкою від заголовків [8].

Стартова строка є обов’язковим елементом, оскільки вказує на тип запиту чи відповіді, заголовки і тіло повідомлення можуть бути відсутніми. Стартові строки для запиту та відповіді відрізняються:

1. запит – Метод_URI HTTP / Версія_протоколу
   (наприклад: GET /kpi /index.html HTTP/1.1);
2. відповідь – HTTP / Версія Код_Стану [Пояснення]
   (наприклад: HTTP / 1.1 Ok) [5].

Метод представляє собою послідовність з будь-яких символів, окрім знаків розділення та керуючих символів. Він визначає операцію, яку необхідно здійснити над вказаним ресурсом. Специфікація HTTP 1.1 не визначає максимально визначену кількість методів, які можна використовувати. Проте, задля того, щоб слідувати загально визначеним стандартам і зберігати сумісність з максимально широким спектром програмних додатків, використовуються лише декілька, загально визначених методів, сенс яких розкритий в специфікації протоколу. Серед таких методів можна визначити наступні:

– GET – використовується для запиту вмісту вказаного ресурсу. Згідно стандарту HTTP, такі запити вважаються ідемпотентними – багаторазове повторення такого запиту буде призводити до одного й того ж результату, при умові, якщо сам ресурс не змінився за цей час. Така властивість надає можливість кешувати відповіді на запити GET [7].

– HEAD – аналогічний методу GET, за виключенням того, що у відповіді сервера відсутнє тіло. Запит HEAD зазвичай використовується для отримання метаданих та перевірки змінився ресурс чи ні [7].

– POST – використовується для передачі даних до заданого ресурсу, при цьому дані включаються в тіло запиту. На відміну від GET запиту, POST не вважається ідемпотентним, тобто багаторазове виконання POST методу може повертати різні результати. Повідомлення відповіді від сервера на виконання методу POST не кешується [7].

– PUT – використовується для завантаження вмісту тіла запиту на вказаний URI. Якщо по вказаному URI не існувало ресурсу, то сервер створює його і повертає статус 201 (Created). На відміну від POST, при запиті PUT клієнт передбачає, що вміст відповідає ресурсу. При цьому метод POST передбачає, що по вказаному URI буде відбуватись обробка вмісту, що передається. Повідомлення відповіді від сервера на виконання методу PUT не кешуються [7].

– PATCH – те саме, що й PUT, але застосовується лише для фрагменту ресурсу [7].

– DELETE – видаляє вказаний ресурс [7].

– TRACE – повертає отриманий запит так, що клієнт може побачити які зміни зробили проміжні сервери [7].

– LINK – встановлює зв’язок вказаного ресурсу з іншими [7].

– UNLINK – видаляє зв’язок вказаного ресурсу з іншими [7].

Кожен сервер повинен підтримувати хоча б методи GET і HEAD. Якщо сервер не зміг розпізнати метод, він повинен повернути статус 501 (Not implemented).

Версія визначає до якої відповідної версії протоколу HTTP визначається запит. Таке значення вказується як два числа, що розділені між собою крапкою (наприклад: 1.1)

Код стану – три цифри, перша з яких вказує на клас стану. Ці цифри визначають результат виконання HTTP запиту. Наприклад, якщо клієнт виконав запит GET, і сервер надає ресурс з вказаним ідентифікатором, тоді такий стан вказується за допомогою коду 200. Якщо ж сервер має повідомити, що такого ресурсу не існує, він має повернути код 404. Специфікація HTTP 1.1 визначає 40 різних кодів станів, а також допускається розширення протоколу для використання додаткових кодів [8].

Пояснення коду стану – текстове, використовується для сприйняття стану людиною, призначене для простого пояснення коду для розробника. Пояснення може не враховуватись клієнтським програмним забезпеченням, а

може відрізнятись від стандартного в деяких визначених реалізаціях серверного програмного забезпечення.

Щодо захисту інформації що передається, то HTTP протокол не передбачає використання шифрування. Проте, існує розширення протоколу, яке використовує упакування даних, що передаються в криптографічний протокол SSL чи TLS. Таке розширення називається HTTPS (HyperText Transfer Protocol Secure). Для HTTPS з'єднань зазвичай використовують TCP-порт 443. На даний момент HTTPS підтримується всіма популярними веб-браузерами такими як Google Chrome, Mozilla Firefox, Safari, Microsoft Edge, тощо.

1.4. REST архітектура

REST (Representational state transfer) – це стиль архітектури програмного забезпечення для розподілених систем (наприклад: World Wide Web), який використовується, зазвичай, для побудови веб-сервісів. Термін REST був введений Роєм Філдингом після публікації своєї докторської дисертації в 2000 році. Варто зауважити, що Рой Філдинг є одним із засновників HTTP-протоколу [9].

Взагалі REST є дуже простим інтерфейсом управління інформацією без використання будь-яких додаткових проміжних шарів. Будь-яка одиниця інформації визначається глобальним ідентифікатором. Таким ідентифікатором є URL. Кожен URL, в свою чергу, має визначений формат. REST – це архітектурний стиль, або множина обмежень для побудови розподілених програмних додатків [9].

Основним поняттям в REST є ресурс. При чому будь-яка інформація, якій можна дати ім'я – є ресурсом. Наприклад: картинка, документ, колекція інших ресурсів, невіртуальний об'єкт, тощо. Ресурс представляє співставлення з набором об'єктів, а не з об'єктом, що відповідає відображенню в будь-який конкретний момент часу. Варто зауважити, що набір станів ресурсів – це і є стан програмного додатку.

Під представленням можна розуміти JSON, чи HTML, чи XML, чи текст в якомусь визначеному форматі, або ж що завгодно, що дозволяє розуміти стан ресурсу та його модифікувати. Представлення, яке модифікує стан ресурсу та представлення, що говорить нам про стан не обов'язково повинні відповідати один одному. REST архітектура розвивалась разом з HTTP 1.1, що був розроблений в 1996-1999 роках.

Згідно принципів SOLID кожен програмний додаток повинен бути продуктивним та легко масштабованим для забезпечення великої кількості компонентів та взаємозв'язку між ними. Саме такими властивостями наділена архітектура REST завдяки тому, що її властивості залежать від накладених на неї обмежень. Такими обмеженнями є:

1. Модель клієнт-сервер. Розмежування обов'язків являється принципом, що лежить в основі цього обмеження. Відокремлення графічного інтерфейсу клієнтської частини від серверної, яка обробляє, аналізує та зберігає дані – підвищує переносимість клієнтської частини коду на різні платформи, а серверну частину додатку робить більш лаконічною та менш завантаженою, що покращує масштабованість [9].

2. Кешування. Клієнтська частина додатку може виконувати кешування даних, що отримуються від серверу, а саме тимчасове збереження на випадок відмови серверу чи багаторазового доступу до одного й того ж ресурсу за визначений час. Сервер в свою чергу повинен надавати явне або неявне позначення того закешований запит чи ні. Це необхідно для того, щоб клієнт не отримував застарілих чи недійсних даних. Правильне використання та реалізація кешування надає змогу покращити взаємодію між клієнтом і сервером, що в свою чергу надає змогу покращити продуктивність та масштабованість всього програмного продукту [9].

3. Шари. Клієнт, зазвичай, не має змогу точно визначити взаємодіє він з напряму сервером чи через проміжні ланки системи. Це обумовлено ієрархічною структурою мережі. Використання проміжних серверів надає змогу балансування навантаження і розподіленого кешування між серверами,

що дає змогу підвищити продуктивність та масштабованість системи. Проміжні ланки також використовуються для забезпечення конфіденційності інформації що передається [9].

4. Одноманітність інтерфейсу. Фундаментальною вимогою проектування REST-сервісів є наявність уніфікованого інтерфейсу. Такі інтерфейси надають можливість кожному з сервісів розвиватись незалежно один від одного. Всі ресурси ідентифікуються в запитах, наприклад, за допомогою використання URI в системах. Ресурси принципово відокремлені від представлень, що повертаються клієнтам. Наприклад, клієнт отримує дані у вигляді HTML, XML, JSON, проте це не означає, що ці дані зберігаються в базі даних саме у такому вигляді. Якщо клієнт володіє метаданими представлення, тоді він має достатньо інформації для того, щоб модифікувати або видаляти ресурс з серверу. Кожне повідомлення має достатньо інформації для того, щоб зрозуміти яким чином його необхідно обробляти [9].

5. Відсутність стану. Протокол взаємодії між клієнтом та сервером потребує виконання наступної умови: в період між запитами клієнта ніяка інформація про його стан не має зберігатись на сервері. Всі запити від клієнта повинні бути створені так, щоб сервер отримав всю необхідну інформацію для виконання таких запитів. Стан сесії при цьому зберігається на стороні клієнту. Під час обробки клієнтських запитів вважається, що клієнт знаходиться в перехідному стані. При тому кожен такий стан програмного додатку представлено зв’язками, що можуть бути використані при наступному зверненні клієнта до сервера [9].

6. Код за вимогою (опціонально). REST може розширити функціональність клієнта за рахунок завантаження коду з сервера на клієнт у вигляді апплетів чи сценаріїв. Рой Філдинг вважає, що таке додаткове обмеження дозволяє проектувати архітектуру, яка підтримує необхідну функціональність в загальних випадках, але, можливо, за виключенням деяких контекстів [9].

Рой Філдинг вважає, що додатки, які не слідують вищеописаним вимогам не можуть називатись REST-додатками [9]. Якщо все ж таки всі вимоги виконуються, тоді такий додаток отримає ряд переваг таких як: надійність, продуктивність, масштабованість, прозорість системи взаємодії, простота інтерфейсів, портативність модулів та компонентів, простота внесення змін. Схематичне зображення роботи REST-архітектури зображено на рис 1.4.

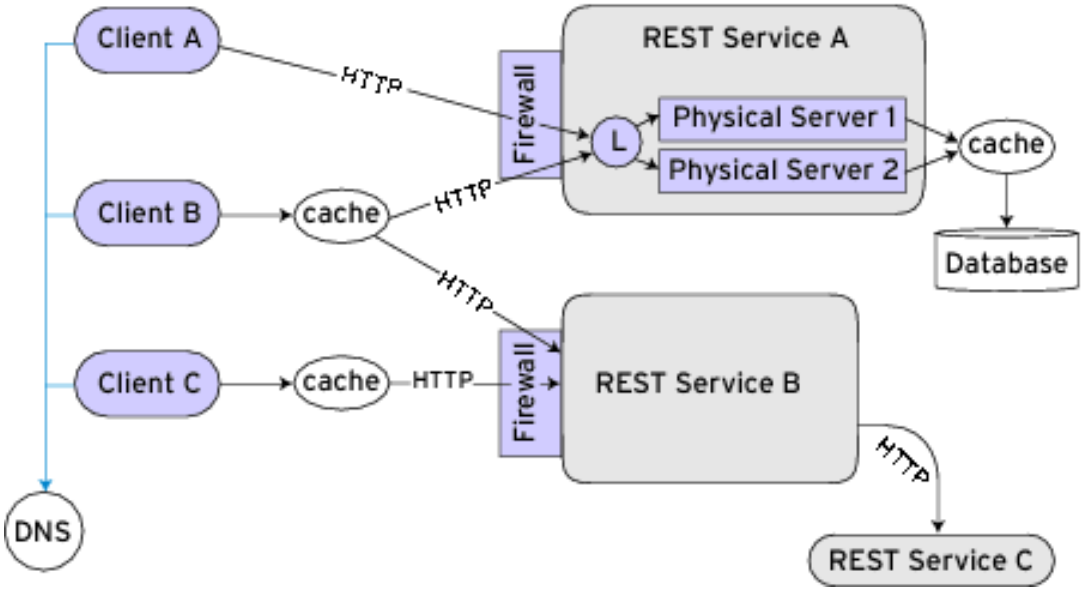

Рисунок 1.4 – Графічне відображення REST архітектури

1.5. Відмінність роботи та використання сокетів та REST архітектури

При побудові надійних та коректних програмних додатків необхідно правильно визначати архітектуру передачі даних між процесами. Варто розуміти, що в різних випадках доцільно використовувати або REST архітектуру, або сокети. Розглянемо два випадки:

1. Коли клієнтському додатку необхідно отримати якісь дані, спільні для інших клієнтів, або ж відправити локальні на сервер, при чому такі дані можуть пройти довгий етап обробки, тоді доцільно використовувати REST архітектуру. В такому випадку клієнт відправляє запит до сервера тільки тоді, коли йому це необхідно. Такий підхід є однозначно ефективним, оскільки сервер не опрацьовує ніяких надлишкових даних від клієнтів, а значить

працює швидше і з меншим навантаженням на апаратне забезпечення. Як приклад можна навести отримання списку репозиторіїв з системи контролю версій. В такому випадку обмін даними між процесами вважається однонаправленим (процеси мають напівдуплексний зв’язок), оскільки сервер надсилає дані до клієнта лише тоді, коли останній запитує на це.

2. Коли серверу необхідно надіслати дані до визначених клієнтів, а останнім в свою чергу до сервера асинхронно, тоді доцільно використовувати сокети. В такому випадку клієнт може відправляти дані до сервера тоді, коли це йому необхідно, але при цьому він матиме змогу асинхронно опрацьовувати дані, що приходять від сервера. Як приклад можна навести будь-який соціальний додаток з можливістю обміну повідомленнями у режимі реального часу. В такому випадку обмін даними між процесами вважається двонаправленим (процеси мають повний дуплексний зв’язок) [4], оскільки як сервер так і клієнт має змогу відправлення даних тоді, коли йому це треба.

Отже, основними відмінностями даних двох підходів є те, що:

1. HTTP, який використовується в REST архітектурі є однонаправленим протоколом, коли запит завжди ініціюється клієнтом, сервер обробляє такий запит і повертає дані, після чого клієнт їх споживає. На відміну від HTTP, сокети мають двонаправлений протокол, де не існує таких визначень, як запит та відповідь. І сервер і клієнт можуть одночасно слати повідомлення один одному.

2. Сокети мають повний дуплексний зв’язок, при якому у визначений час обмін даними між клієнтом та сервером проходить одночасно, незалежно один від одного.

Типово, що один TCP зв’язок утворюється для запиту HTTP і завершується тоді, коли відповідь повертається. Новий TCP зв'язок утворюється для кожного циклу запит\відповідь. При використанні сокетів такої проблеми не виникає. В такому випадку створюється лише одне TCP з’єднання, яке використовується впродовж всієї сесії. Саме тому швидкість

обміну даними між процесами за допомогою сокетів є швидшим ніж за допомогою HTTP запитів.

В даній магістерській дисертації розробляється програмна бібліотека, яка дозволяє побудувати легко масштабований програмний додаток, який може обмінюватись даними з іншими програмними додатками за допомогою сокетів. Програмування на транспортному рівні досить складне та трудомістке, оскільки воно не є тривіальним. За допомогою програмної бібліотеки розробленої в даній магістерській дисертації вирішується не лише проблема складності реалізації, а й пропонується легке розширення функціональності, а також керування потоками змін. Це забезпечується за допомогою парадигми реактивного програмування.

На рис. 1.5 зображено графічно порівняння вищеописаних двох технологій.

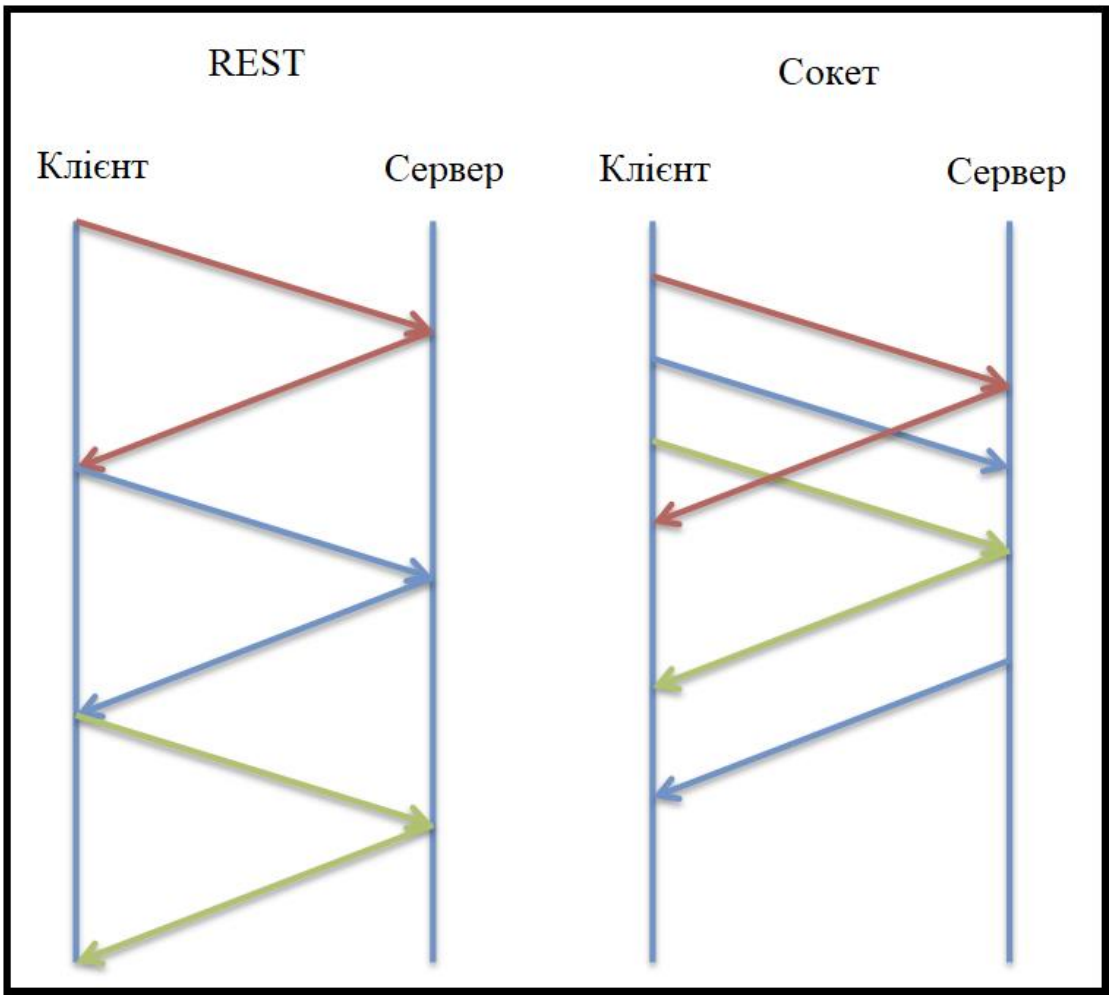

Рисунок 1.5 – Графічне порівняння роботи сокетів та REST архітектури

2. МЕТОДИ РЕАЛІЗАЦІЇ ПРОГРАМНОЇ БІБЛІОТЕКИ

Метою створення програмної бібліотеки є

- полегшення в плані використання;
- простота розробки обміну даними між процесами;
- збільшення швидкодії передачі даних;
- збільшення абстракції передачі даних на транспортному рівні;
- покращення легкості сприйняття програмного коду;
- можливість простого масштабування програмних додатків.

Для того, щоб забезпечити таку немалу кількість критеріїв, необхідно підходити до проблеми комплексно. Тому в першу чергу необхідно обрати архітектуру, яку використовуватиме та пропонуватиме програмна бібліотека. Для цього була обрана парадигма реактивного програмування. Також, важливим є вибір мови програмування, на якій написана бібліотека, оскільки це впливає на універсальність її використання.

2.1. Парадигма реактивного програмування

На перший погляд поняття реактивного програмування не є цілком інтуїтивним, тому, перш за все, необхідно дати йому визначення.

Реактивне програмування – парадигма, що автоматизує розсилання змін через потоки даних. Це є програмування з асинхронними потоками даних. Вже це визначення дає розуміння того, що саме це необхідно при розробці обміну даними між процесами, оскільки за допомогою такого підходу вирішується велика проблема автоматичного розсилання змін представлення при отриманні нових даних [10].

За останні декілька років вимоги до програмних додатків значно підвищились. Лише декілька років назад великі програми мали по 10 серверів, секунди для обробки одного запиту, години оффлайн підтримки та гігабайти даних. Сьогодні ж програмні додатки зосереджені будь-де: в смартфонах, розумних будинках, кавоварках, а також кластерах, що працюють на хмарному середовищі та підтримують тисячі мультиядерних процесорів. Нині

користувачі вимагають, щоб запити виконувались мілісекунди та мали 100% безвідмовної роботи. Дані вимірюються в петабайтах. Сьогоднішні вимоги просто не дотримуються вчорашніх програмних архітектур.

Необхідний узгоджений підхід до системної архітектури, а всі аспекти вже визнані індивідуально. Потрібно, щоб системи відповідали таким вимогам: були чутливими, стійкими, еластичними та орієнтованими на повідомлення. Такі системи називаються реактивними.

Системи, побудовані як реактивні, є більш гнучкими, менш зв'язаними та більш масштабованими. Це робить їх легшими для розвитку і змін. Реактивні системи дуже чутливі, що дає користувачам ефективний інтерактивний відгук.

Отже, для того, щоб система була реактивною, необхідно, щоб вона відповідала таким вимогам:

– **Чутливість**. Система має відповідати своєчасно, якщо це можливо. Чутливість є основним фактором легкого використання та корисності програмного продукту, але більше того, чутливість означає, що проблеми можуть бути швидко виявлені та ефективно працювати. Ресурси системи зосереджені на забезпеченні швидкого і постійного часу відгуку, встановленні надійної верхньої межі, що забезпечує постійну якість обслуговування. Ця послідовна поведінка, у свою чергу, спрощує обробку помилок, створює довіру кінцевих користувачів та заохочує подальшу взаємодію з ними.

– **Стійкість**. Система залишається чутливою в умовах аварії. Це стосується не лише високонавантажених, критично важливих систем – будь-яка система, яка не є стійкою, після невдачі не буде реагувати. Стійкість досягається реплікацією, стримуванням, ізоляцією та делегуванням. Ізолюючи компоненти один від одного гарантується, що частини системи можуть вийти з ладу та відновитись, не посягаючи на систему в цілому. Відновлення кожного компонента делегується на інший (зовнішній) компонент, а висока доступність забезпечується реплікацією, де це необхідно. Клієнт компонента не залежний від обробки його помилок.

– **Еластичність**. Система залишається чутливою при різному навантаженні. Реактивні системи можуть реагувати на зміни рівня введення даних, збільшуючи або зменшуючи ресурси, призначені для обслуговування цих даних. Це визначає системи, які не мають центральних вузьких місць, в результаті чого можна відокремити або відтворити компоненти та розподіляти ресурси серед них. Реактивні системи підтримують алгоритми прогнозування, а також реактивні масштабування, забезпечуючи релевантні показники живого виконання. Вони досягають еластичності економічно ефективним способом на товарних апаратних і програмних платформах.

– **Орієнтовність на повідомлення**. Реактивні системи покладаються на асинхронну передачу повідомлень для того, щоб встановити межу між компонентами, що забезпечує втрату зчеплення, ізоляцію та прозорість розташування. Ця межа, також, забезпечує засоби передачі відмов як повідомлень. Використання явного передавання повідомлень дозволяє керувати навантаженням, еластичністю та потоком, формуючи та контролюючи черги повідомлень у системі та застосовуючи при необхідності зворотний тиск (Backpressure). Неблокуюча комунікація дозволяє клієнтам споживати ресурси лише під час активної роботи, що призводить до зменшення накладних витрат системи.

Графічне відображення взаємозв’язку вищеописаних вимог зображено на рис. 2.1.

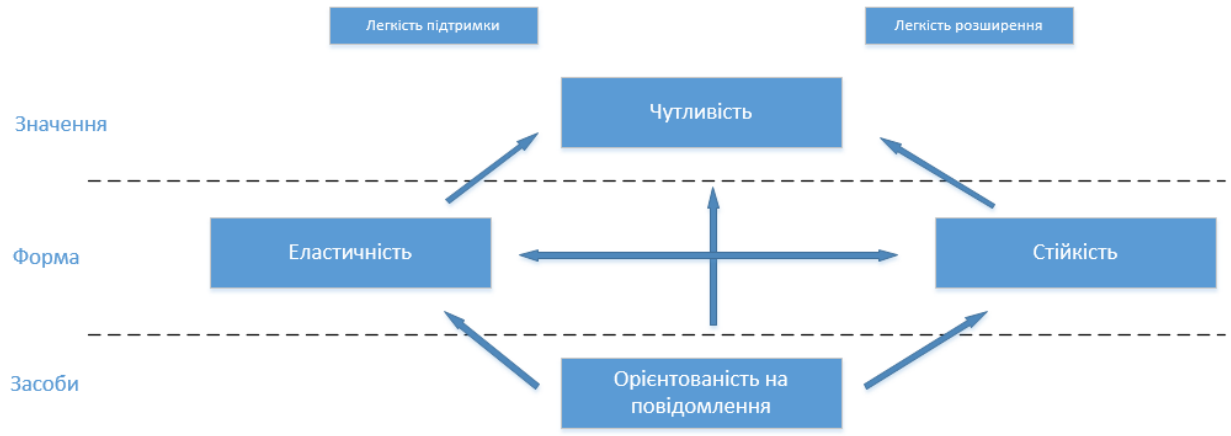

Рисунок 2.1 – Взаємозв’язок загальних вимог реактивних систем.

Великі системи складаються з менших і тому залежать від реактивних властивостей їх складових. Це означає, що реактивні системи застосовують принципи дизайну, тому ці властивості застосовуються на всіх рівнях масштабу, що робить їх складними. Найбільші системи у світі покладаються на архітектуру, засновану на цих властивостях, і служать потребам мільярдів людей щодня. Варто застосовувати ці принципи проектування свідомо з самого початку, а не повторно відкривати їх кожного разу.

Декларація реактивного програмування дуже важлива, оскільки вона задає архітектуру побудови реактивних програмних додатків. Проте для побудови важлива не лише архітектура, а й її конкретна реалізація.

В 2010 році Ерік Майєр (Erik Meijer) співробітник компанії Microsoft запропонував модель реактивних розширень (Reactive Extensions) та розробив його конкретну реалізацію, як набір бібліотек – Rx.NET для обробки асинхронних потоків даних. Такі потоки побудовані на подіях, наприклад, при натисненні на кнопку відбуваються якісь обчислення. В інакшому випадку після натисненні на кнопку довелось би періодично перевіряти відбулась подія чи ні і тільки після успішного завершення виконувати обчислення. Проте, в такому підході значно зростає навантаження на сервер, оскільки може бути дуже багато періодичних звернень від різних користувачів.

В 2012 році компанія Microsoft виклала сирцевий код бібліотеки Rx.NET у відкритий доступ. Після чого інші розробники написали подібні бібліотеки для інших мов програмування: RxCpp, RxJS, RxPHP, RxJava, Rx.rb, Rx.py, RxKotlin, RxSwift, RxScala.

Перечислені бібліотеки містять не лише основну функціональність, яка притаманна парадигмі реактивного програмування, а й значну кількість операторів, за допомогою яких можна змінювати, фільтрувати, видаляти, отримувати дані з певного визначеного потоку. Потоки можуть бути використані як вхідні аргументи інших потоків. Можна поєднати два потоки, та визначити порядок, в якому будуть отримані дані з цих двох потоків.

Оскільки потік є центральним параметром реактивного програмування, тоді необхідно визначити що це таке. Потік – послідовність, що складається з постійних подій, відсортованих в часі. В ньому може бути три типи подій: значення (дані визначеного типу), помилка, сигнал про завершення роботи. Це зображено на рис. 2.2.

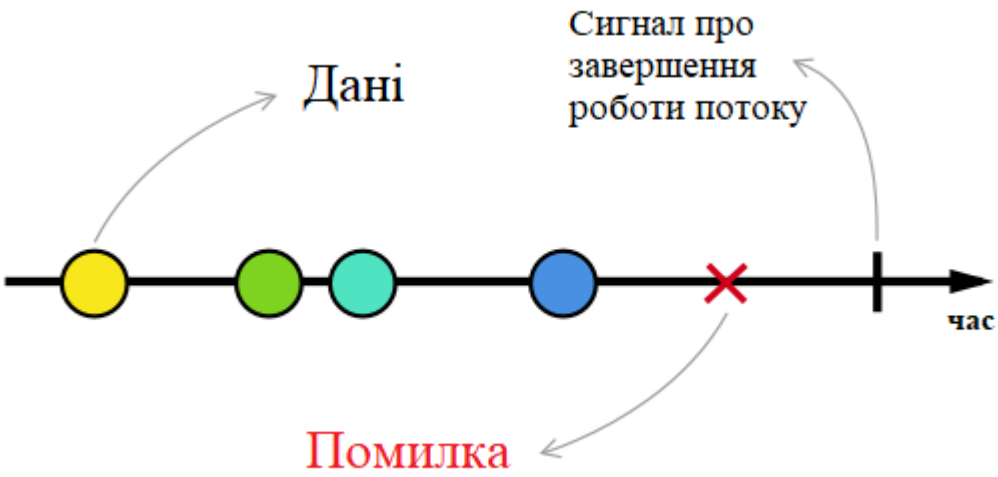

Рисунок 2.2 – Графічне відображення роботи потоку в парадигмі реактивного програмування.

Згенеровані дані отримуються завжди асинхронно. Вся парадигма реактивного програмування побудована завдяки шаблону проектування «спостерігач».

Спостерігач – поведінковий шаблон проектування. Він створює такий механізм, при якому одні об’єкти можуть слідкувати за подіями, які відбуваються на інших об’єктах. Такий шаблон використовує відношення «один до багатьох». В такому відношенні існує один об’єкт, за яким спостерігають багато спостерігачів. При зміні спостережуваного об’єкту автоматично відбувається сповіщенні спостерігачів [11].

Основними елементами такого шаблону проектування є «підписник (Subscriber)» та «видавець (Publisher)». Паттерн пропонує зберігати список посилань на об’єкти підписників в об’єктах видавець. В такому випадок кожен видавець знатиме яких підписників йому треба сповіщати при оновленні даних. Також, об’єкт видавець містить методи для додавання нового

підписника та видалення існуючого. Такий взаємозв’язок зображено на рис. 2.3.

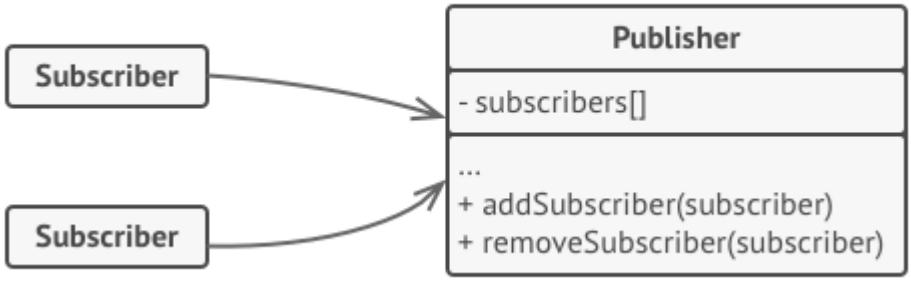

Рисунок 2.3 – Схема відображення взаємозв’язку підписника з видавцем.

Основною задачею видавця є сповіщення, тому такий метод теж реалізований в об’єкті видавця. Варто зауважити, що кожен об’єкт спостерігача повинен слідувати загальному інтерфейсу, який містить метод сповіщення спостерігача. В такому випадку видавцеві байдуже яка конкретна реалізація користувача, оскільки його задача просто викликати метод сповіщення. Це зображено на рис. 2.4.

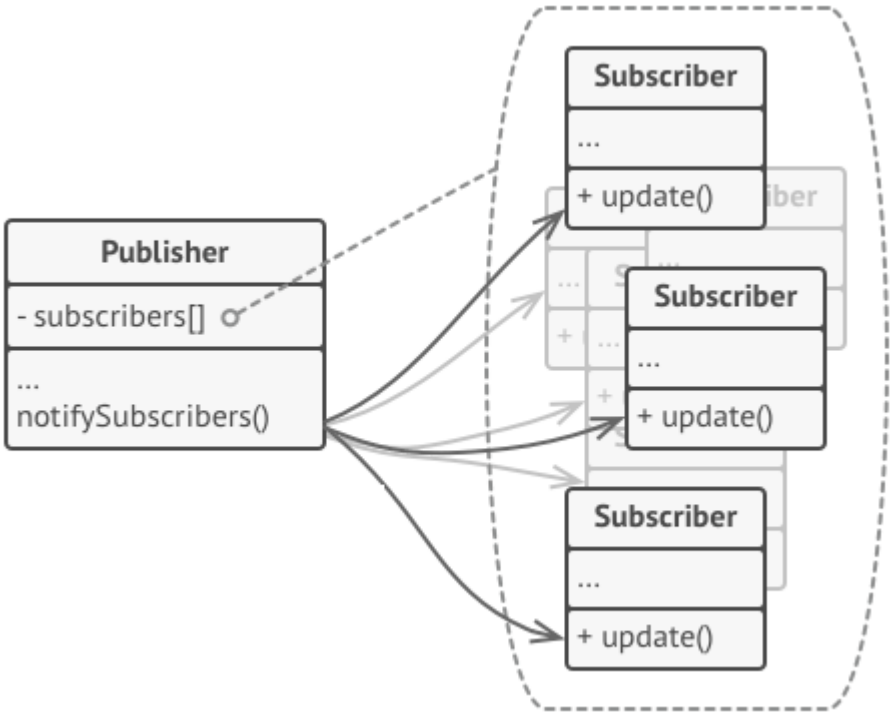

Рисунок 2.4 – Сповіщення спостерігачів видавцем.

Отже, структура роботи шаблону проектування «спостерігач» така:

1. **Видавець** містить внутрішній стан, дані та події, які можуть змінюватись з плином часу. Окрім того, об’єкт видавця містить список об’єктів підписників, а також методи для їх додавання та видалення [11].

2. Коли внутрішній стан, дані чи події видавця змінюються, тоді він сповіщує всіх підписників, що належать на цьому видавцю. Завдяки узагальненому інтерфейсу не виникає проблеми в тому щоб правильно сповістити кожного підписника, а ті, в свою чергу, можуть по різному обробляти сповіщення [11].

3. **Підписник** визначає інтерфейс з одним методом, яким користується видавець для надсилання сповіщень [11].

4. **Конкретні підписники** оброблюють сповіщення для кожного видавця по різному, оскільки мають різну реалізацію. Вони мусять реалізовувати узагальнений інтерфейс «Підписник», щоб видавець не залежав від конкретних класів підписників [11].

5. Після того, як було отримане сповіщення, підписнику треба отримати оновлені дані, події чи стан видавця. Видавець може передати такі дані через параметри методу сповіщення. Підписник може зберігати посилання на екземпляр видавця, переданий йому через аргумент конструктора [11].

6. **Клієнт** створює об’єкти підписників і видавців, а потім реєструє підписників на оновлення у видавцях [11].

Загальну структуру паттерну «спостерігач» зображено на рис. 2.5. Кроки реалізації шаблону проектування «спостерігач»:

1. Розділити функціональність на дві частини: основна незалежна частина та опціональні залежні частини. Основна незалежна частина стане видавцем, а залежні частини стануть підписниками.

2. Створити інтерфейс підписників. Достатньо буде визначити тільки один метод - сповіщення.

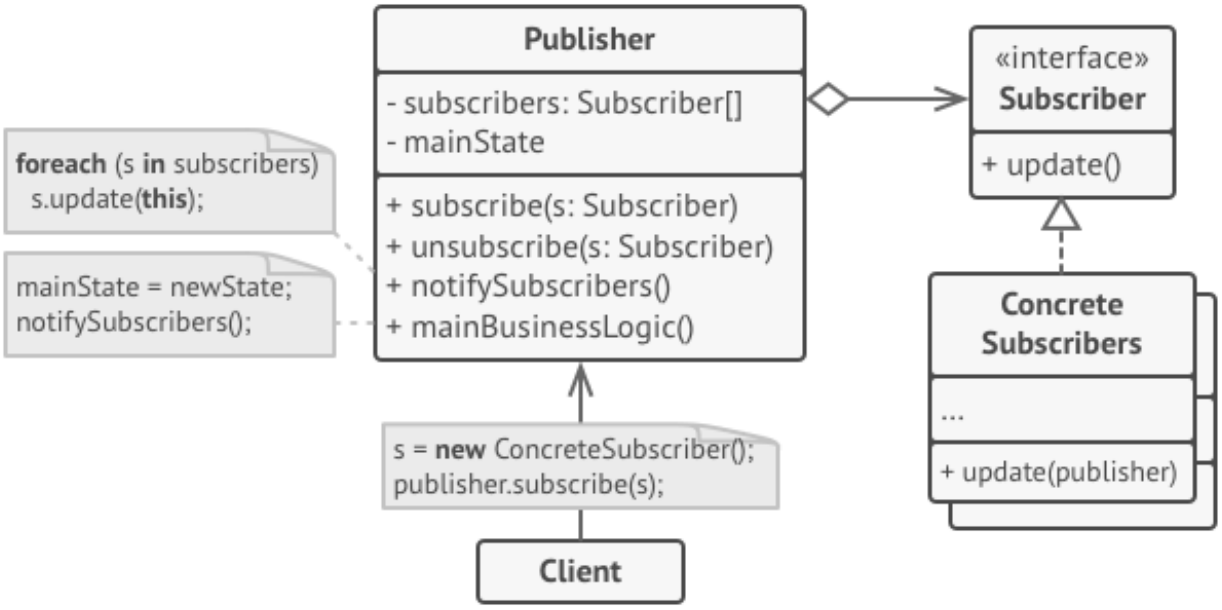

Рисунок 2.5 – UML-діаграма шаблону проектування «спостерігач»

3. Створити інтерфейс видавців та описати у ньому операції управління підпискою. Видавці повинні працювати з підписниками лише через загальний інтерфейс.

4. Створити класи конкретних видавців. Реалізувати їх таким чином, щоб після кожної зміни стану вони слали сповіщення всім своїм підписникам.

5. Реалізувати метод сповіщення в конкретних підписниках. Не забути передбачити параметри, через які видавець міг би відправляти дані, пов’язані з подією, що відбулась. Можливий і інший варіант, коли підписник, отримавши сповіщення, сам візьме потрібні дані з об’єкта видавця. Але в цьому разі ви будете змушені прив’язати екземпляр класу підписника до конкретного класу видавця.

6. Клієнт повинен створювати необхідну кількість об’єктів підписників та підписувати їх у видавців [11].

В цілому такий підхід звільняє розробників від написання купи непотрібного програмного коду для підтримки станів, замість того щоб керувати станами необхідно просто зв’язати компоненти один з одним. Якщо брати до уваги парадигму реактивного програмування, то там всі поняття є

тотожними з шаблоном проектування «спостерігач», проте є дуже багато розширення функціональності і невелика різниця в назвах.

Оскільки програмна бібліотека, що розробляється в даній магістерській дисертації, писатиметься на мові програмування Kotlin, тоді для опису основних методів парадигми реактивного програмування візьмемо до уваги її конкретну реалізацію – RxJava 2. Для аналізу було обрано саме цю бібліотеку, оскільки мова програмування Kotlin виконується на віртуальній машині Java (JVM).

RxJava пропонує:

– набір класів для представлення джерел даних;
– набір класів для прослуховування джерел даних;
– набір методів для перетворення та комбінування даних (оператори).

Джерело даних виконує деяку роботу, коли починається або закінчується його прослуховування. Джерело може працювати як синхронно так і асинхронно, видавати як багато елементів, так і може бути пустим. Джерело може завершити свою роботу успішно, може впіймати помилку та опрацювати її згідно правил встановлених розробником. Також, джерело може працювати нескінченно, поки працює програма. В RxJava, на відміну від звичайного шаблону проектування «спостерігач» існують різні види джерел даних:

– Observable – джерело даних, яке може видавати будь-яку кількість елементів. Таке джерело може завершуватись успішно, або ж з помилкою. При успішному завершенні викликається метод onSuccess(), коли приходить наступний елемент викликається метод onNext(T element), коли виникла помилка при видачі наступного елементу, викликається метод onError(Throwable t);

– Flowable – джерело даних, наділене такими ж властивостями, що й Observable, проте має підтримку зворотнього тиску (Backpressure). Backpressure – момент, коли джерело даних видає елементи швидше, ніж підписник може їх прийняти. Для уникання таких колізій існують методи: onBackPressureDrop() – елементи, які не можуть бути опрацьовані

підписником будуть пропущені, onBackpressureBuffer() - елементи, які не можуть бути опрацьовані підписником будуть записані в буфер та опрацьовані пізніше.

– Single – джерело даних, що має такі ж властивості, як і Observable, проте може видати лише один елемент.

– Maybe – джерело даних, що має такі ж властивості, як і Observable, проте може видати один елемент, або ж не видати ні одного.

– Completable – джерело даних, що має такі ж властивості, як і Observable, проте не видає елементи, та не містить методу onNext(T element), оскільки він лише демонструє відбувся запит успішно чи ні.

Окрім джерел даних в RxJava існують такі компоненти:

– Observer – об'єкт чи функція, що визначає як необхідно опрацьовувати дані, що приходять від джерела;

– Subscriber – об'єкт чи функція, що пов'язує Observable та Observer.

Графічне відображення роботи всіх компонентів реактивного програмування зображено на рис. 2.6.

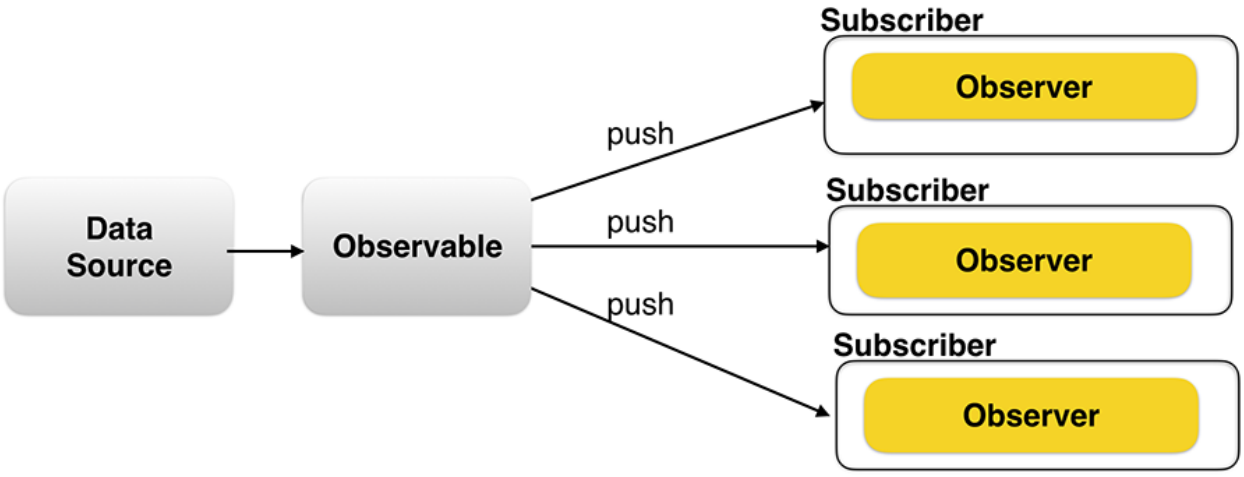

Рисунок 2.6 – Відображення роботи компонентів парадигми реактивного програмування в реалізації RxJava.

Можна з легкістю аналізувати та опрацьовувати дані між моментом, коли Observable їх відправив та моментом коли підписник їх отримав. Це реалізовано за допомогою операторів. Кожен такий оператор – це функція, що

приймає на вхід Observable, трансформує отримане значення і видає новий Observable на виході. Так як вхід та вихід будь-якого оператору являється одного і того ж типу, їх можна зв’язувати. Оператори можна розділити на такі категорії:

– створення (from, just, range, interval, fromCallable, fromRunnable, fromIterable);

– перетворення (map, flatMap, switchMap, concatMap);

– комбінуючі (merge, withLatest, zip, combineLatest);

– фільтруючі (filter);

– обробки помилок (onExceptionResumeNext, onErrorResumeNext, onErrorReturn, retry, retryWhen);

– математичні (reduce, collect)

– умовні та булеві (amb, contains, skipUntil, skipWhile, takeUntil, takeWhile, all);

В RxJava існує міст між Observer та Observable, він називається Subject. Він може працювати як Observer, так і Observable. Він може підписатись на один чи більше видавців, а також може пройти через всі елементи, за якими він спостерігає. Більше того, Subject може видавати нові елементи. Серед конкретних реалізацій Subject необхідно виділити наступні:

– Behaviour Subject – видає попередній елемент та всі наступні, які будуть після підписки на нього;

– Publish Subject – видає всі елементи, які надійдуть після підписки;

– Async Subject – видає останній елемент і тільки останній;

– Replay Subject – видає всі елементи, які містить, незалежно від того коли на нього підписався підписник;

Завдяки RxJava також можна вказувати на якому потоці виконувати запит до бази даних, а на якому відображати отримані дані. В такому випадку можна виконувати досить багато задач на різних потоках, що значно пришвидшить виконання програмного додатку вцілому.

2.2. Обґрунтування використання мови програмування Kotlin

Компанія з Санкт-Петербургу з назвою JetBrains розпочала розробку своєї мови програмування Kotlin в 2010 році. Офіційний реліз продукту був випущений в 2016 році. Таку назву мова отримала в честь острова в Фінській затоці, на якому розташований Кронштадт. Як повідомляється в прес-релізі, Kotlin повинен працювати скрізь, де працює Java, і один з орієнтирів був зробити такий продукт, який можна буде використовувати в змішаних проектах, які створюються на декількох мовах [12].

Як і Java, C і C ++, Kotlin - це статично типізована мова. Вона підтримує як об'єктно-орієнтоване, так і процедурне та функціональне програмування. На конференції Google I/O 2017 команда розробників Android повідомила, що Kotlin отримав офіційну підтримку для розробки Android-додатків, що надало мові програмування неймовірної популярності в світі [12].

Kotlin має значний приріст розробників, що використовують його за останній рік, майже вдвічі більше, ніж в 2017 році, і втричі більше ніж в 2016 році [12]. Це зображено на діаграмі 2.7.

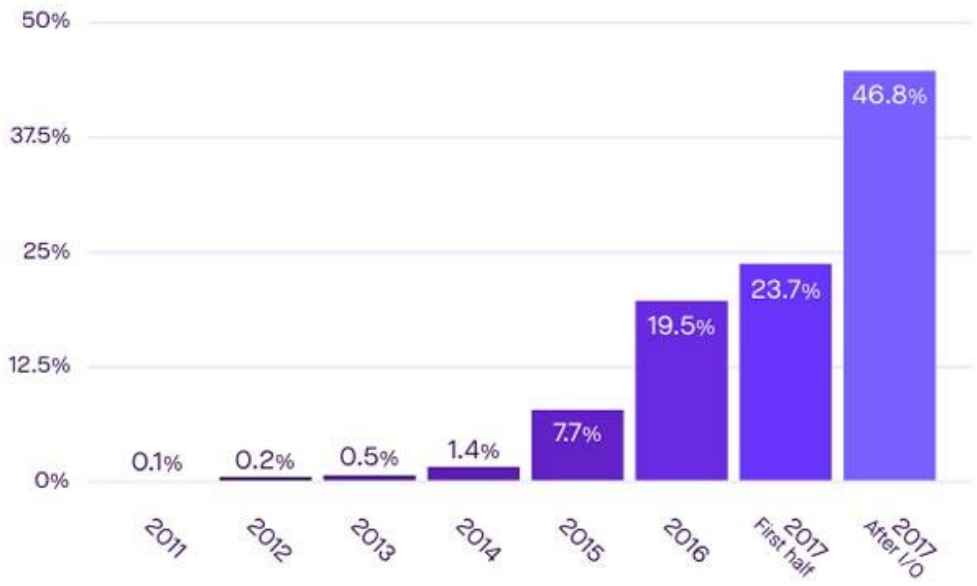

Рисунок 2.7 – Порівняльна діаграма кількості розробників, що використовують мову програмування Kotlin в залежності від року.

Як відзначають автори Kotlin, найголовніше для них було створити «прагматичний» продукт. Це означає, що вони фокусувалися не лише на усуненні помилок Java і вдосконаленні продукту, що робив би будь-який програміст-розробник, а хотіли зробити саме корисний інструмент [12].

Дві головні особливості Kotlin, це його простота і повна сумісність з Java. Kotlin створювався компанією, яка робить дуже багато продуктів на Java і яка добре розбирається в сучасних інструментах розробки. Запит на нову мову існує давно, але зробити таку мову, яка б дозволила взяти величезну готову кодову базу Java, надати новий інструмент і без перешкод продовжувати розробку з більшою ефетивністю - такого інструменту до появи Kotlin не існувало. Творці нової мови дуже добре відчули потреби бізнесу і розробників: бізнесу дали можливість збільшити ефективність розробників, а розробникам дати сучасний інструмент для розробки. При тому, творці надали не лише компілятор, а й підтримку в IDE, які підкреслює переваги нової мови програмування [12].

Дуже важливим моментом є те, що Kotlin має підтримку Java 1.6, адже саме ця версія Java використовується у всіх сучасних версіях Android, і, не дивлячись на запланований перехід на OpenJDK, восьма версія потрапить в руки розробникам під мобільні пристрої не так скоро як хотілося б. При цьому, збільшення розміру кінцевого програмного додатку не є суттєвим як на теперішній час: 823 Кб (для версії 1.0.0). Хоч Kotlin і підтримує версію Java 1.6, проте він містить всі особливості, які має Java 1.8 і навіть більше. Завдяки такій мові програмування розробка програмних додатків пришвидшується в рази [12].

Особливості мови програмування Kotlin, що надають перевагу перед іншими мовами:

1. **Null Safety (захищеність від Null).** Вважається основною і найбільш популярною перевагою перед Java. Перевірка на ти, що може бути Null перевіряється на етапі компіляції [13]. В такому випадку виключаються всі випадки завершення програмного додатку з помилкою на етапі його

виконання. Для сумісності з Java були створені анотації @Nullable та @NotNull, які позначають для компілятора являється даний тип nullable чи ні. Для того, щоб вказати, що тип може бути Null необхідно поставити символ «?» після типу. Наприклад, у випадках, коли дані отримуються з джерела і невідомо чи може прийти значення Null чи ні. Саме тому це дуже важлива особливість при обміні даними між процесами. Для перевірки змінної на null досить використати знак «?», наприклад:

```
val p: String? = null
print("${p?.get(0)}")
```

В даному прикладі не виникне помилки при виконанні програми, оскільки відбулась перевірка на Null. На екран виведеться надпис «null».

В Kotlin існує оператор, який називається Елвіс-оператором. Він створює значення за замовчуванням заміть Null, наприклад:

```
val p: String? = null
print("${p?.get(0) ?: "There was a problem"}")
```

В даному прикладі також не виникне помилки при виконанні програми, оскільки відбулась перевірка на Null, проте на екран виведеться строка «There was a problem».

Також, існує функція let, яка виконує функцію лише в тому випадку, якщо змінна не є Null, наприклад:

```
val p: String? = null
p?.let {
    print(p)
}
```

В даному прикладі також не виникне помилки при виконанні програми, оскільки відбулась перевірка на Null, проте на екран не виведеться нічого, оскільки функція не виконалась.

Not-null assertion явно вказує, що цей тип в місці виклику оператора не є Null, в інакшому випадку відбудеться аварійне завершення роботи програмного додатку, наприклад:

```
val p: String? = null
print("${p!!.get(0)}")
```

В даному прикладі відбудеться помилка KotlinNullPointerException.

2. Розширюючі методи. Kotlin, подібно до C # і Gosu, надає можливість розширювати клас з новою функціональністю без наслідування з класу або використовувати будь-який тип шаблону дизайну, наприклад, декоратор [13]. Це робиться за допомогою спеціальних декларацій, що називаються розширеннями. Kotlin підтримує розширення як функцій, так і властивостей. Така особливість мови Kotlin використовується в багатьох інших аспектах даної мови програмування, наприклад, в перевизначенні операторів, чи Kotlin DSL. Розширення насправді не змінюють класи. Визначаючи розширення нові методи не додаються у клас, а просто створюються нові функції, які можна викликати за допомогою символу «.» на змінних цього типу. Варто відзначити, що функції розширення посилаються статично, тобто вони не є віртуальними за типом приймача. Це означає, що викликана функція розширення визначається типом виразу, на який викликається функція, а не типом результату цього виразу під час виконання [13]. Наприклад:

```
open class C
class D: C()

fun C.foo() = "c"

fun D.foo() = "d"

fun printFoo(c: C) {
    println(c.foo())
}

printFoo(D())
```

У цьому прикладі буде надруковано "c", тому що викликана функція розширення залежить тільки від оголошеного типу параметра c, який є класом C.

3. Відсутність примітивних типів. Можна сказати, що це не перевага, а швидше навпаки – недолік, оскільки об’єкти займають більше пам’яті ніж примітиви. Проте, це вимушений хід з боку мови програмування задля того, щоб забезпечувати null safety. Варто зауважити, що примітиви Java

викликаються самим компілятором задля економії пам’яті. Вони викликаються тоді, коли змінна не може бути Null [13].

4. Автоматична генерація коду. Kotlin часто автоматично генерує шаблонний програмний код, наприклад, методи отримання та встановлення значень змінних в класі (getters та setters). Проте, якщо розробнику необхідно не просто отримати значення, а якось його перетворити, тоді є можливість з легкістю перевизначити ці методи. Також, Kotlin генерує автоматично допоміжні методи суперкласу Object, для перевірки об’єктів на рівність, генерування їхнього хеш-коду, виведення об’єкту у вигляді строки та інших методів. Більше того, Kotlin може реалізувати автоматично відомі шаблони проектування, такі як «Синглтон» чи «Делегат» [13]. Для реалізації об’єкту-синглтону необхідно просто написати слово object, наприклад:

```
object singleton
```

Всього два слова і синглтон об’єкт готовий.

5. Лямбда функції. Звичайно, як будь-яка сучасна мова з претензією на можливість функціонального програмування, у Kotlin функція - це сутність першого класу, якщо перекладати дослівно (first-class functions). Тобто функції можна не тільки оголосити прямо в пакеті (в Java їх видно все рівно тільки в класах - по імені файлу), але і передавати як параметри, повертати з інших функцій, тощо [13]. Сьогодні, звичайно, нікого цим не здивувати, проте у порівнянні з Java, де синтаксичних функцій немає (а тільки функціональні інтерфейси), в Kotlin повноцінний синтаксис для оголошення функції:

```
fun passTen(func: (Int)->Int ): ()->Int {
    return { func(10) }
}
```

6. Перевизначення операторів. Kotlin дозволяє забезпечувати реалізацію заздалегідь визначеного набору операторів на всіх типах. Ці оператори мають фіксований символічний вигляд (наприклад, + або *) та фіксовану перевагу. Для реалізації оператора ми надаємо звичайну функцію або функцію розширення з фіксованим ім'ям для відповідного типу. Функції,

що перевизначають оператор повинні бути позначені модифікатором «operator» [13]. Нижче наведено приклад класу Counter, який починається з заданого значення, і його можна збільшити, використовуючи перевантажений оператор «+» :

```
data class Counter(val dayIndex: Int) {
    operator fun plus(increment: Int): Counter {
        return Counter(dayIndex + increment)
    }
}
```

Зображення символічного типу оператора відносно імені для його перевизначення в Kotlin зображено в таблиці 2.1.

1. Вивід типів. В Kotlin не обов'язково вказувати тип змінної, варто лише визначити що це: значення чи змінна. Для визначення змінної пишеться ідентифікатор «var», а для визначення значення – ідентифікатор «val». Наприклад:

```
val valueText = "10" // Значення з автоматичним
визначенням типу - String
var variableText = "12" // Змінна з автоматичним
визначенням типу - String
val valueTextWithType: String = "14" // Значення з явно
визначеним типом - String
var variableTextWithType: String = "16" // Значення з
явно визначеним типом - String
```

Значення змінної «var» можна змінити, а значення значення «val» - ні.

2. Компаньйон-об'єкти (companion objects). У Kotlin, на відміну від Java або C#, класи не мають статичних методів. У більшості випадків рекомендується просто використовувати функції рівня пакету. Якщо необхідно написати функцію, яка може бути викликана без екземпляра класу, але потрібний доступ до внутрішніх елементів класу (наприклад, заводський метод), можна записати його як член декларації об'єкта в цьому класі. Навіть більше, якщо був визначений компаньйон-об'єкт у локальному класі, то можна викликати його поля з таким самим синтаксисом, як виклик статичних методів у Java / C#, використовуючи лише назву класу як кваліфікатор [13].

Таблиця 2.1 – Відношення виразу оператора перевизначення до його методу

Вираз	**Ім’я методу перевизначення**
a += b	a.plusAssign(b)
a -= b	a.minusAssign(b)
a *= b	a.timesAssign(b)
a /= b	a.divAssign(b)
a %= b	a.remAssign(b)
a == b	a?.equals(b) ?: (b === null)
a != b	!(a?.equals(b) ?: (b === null))
a > b	a.compareTo(b) > 0
a < b	a.compareTo(b) < 0
a >= b	a.compareTo(b) >= 0
a <= b	a.compareTo(b) <= 0
a()	a.invoke()
a[i]	a.get(i)
a[i] = b	a.set(i, b)
a[i, j] = b	a.set(i, j, b)
a in b	b.contains(a)
a !in b	!b.contains(a)
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.rem(b)
a..b	a.rangeTo(b)
a++	a.inc()
a--	a.dec()

3. **Розумне приведення типів (Smart casts).** Якщо відбулась перевірка того, що деяка змінна являється визначеним типом, тоді, на наступних кроках виконання програми відбудеться автоматичне приведення цієї змінної до визначеного типу. Наприклад:

```
if (x !is String) return
print(x.length) // x автоматично приведено до строки
```

Завдяки цій особливості зменшується ймовірність неправильно приведення типу, яке часто виникає навіть на великих комерційних проектах.

4. **Вирази діапазонів.** Вирази діапазонів сформовані за допомогою функції rangeTo, які мають форму оператора «..» яка доповнюється операторами «in» та «!in». Діапазон визначається для будь-якого типу, що можна порівняти, проте для інтегрованих примітивних типів він має оптимізовану реалізацію. Ось кілька прикладів використання діапазонів:

```
if (i in 1..10) { // те саме що й 1 <= i && i <= 10
    println(i)
}
```

Також, можна використовувати вирази діапазонів в циклах, при цьому можна задавати крок (step) та напрям (downto) проходу циклу.

Отже, проаналізувавши всі переваги мови Kotlin, можна зробити висновок, що дана мова програмування чудово підходить для розробки програмної бібліотеки обміну даними між процесами засобами реактивного програмування. Kotlin має підтримку функціонального програмування, що підвищує рівень абстракції при реалізації парадигми реактивного програмування. Також, варто відзначити, що кодова база значно зменшується при розробці на Kotlin, що покращує читабельність коду, а також збільшує швидкість реалізації конкретно визначених задач. 100% сумісність коду з мовою програмування Java надає те, що немає потреби переписувати великі програмні додатки з нуля, варто просто додати залежність для підтримки Kotlin .

3. СТРУКТУРА ТА АЛГОРИТМИ РОБОТИ ПРОГРАМНОЇ БІБЛІОТЕКИ

3.1. Внутрішні предметно-орієнтовані конструкції Kotlin DSL

Предметно-орієнтована мова (або domain-specific language) – мова, що вирішує лише вузьконаправлену задачу, наприклад: запит в базі даних, генерація коду, задання конфігурації, опис процесу збірки програмного додатку, тощо [13]. Kotlin не є DSL, проте він містить в собі методи та рішення, за допомогою яких можна написати свою власну внутрішню DSL структуру, яка надасть змогу абстрагувати використання рішення деякої вузької задачі від її конкретної реалізації. За допомогою цього можна з легкістю досягти ефекту «чорного ящика», коли методи реалізації приховані від користувача так, що останньому необхідно задати лише визначену конфігурацію. DSL – це спосіб описати програму чи його алгоритм не в імперативному стилі (як отримати результат), а в декларативному (описати поточну задачу), в такому випадку рішення проблеми буде отримано судячи з заданої інформації.

В даній магістерській дисертації була побудована DSL для того, щоб пришвидшити розробку програмних продуктів, а також абстрагувати низькорівневий код. Все це повинно мати «чистий» програмний інтерфейс. Тому, необхідно визначитись, що значить «чистий». Програмний інтерфейс вважається «чистим», якщо він задовольняє такі умови:

– те, що відбувається в програмному коді повинно бути чітким для читачів. Це можна досягти вірним вибором імен класів, полів та змінних, що важливо на будь-якій мові;

– програмний код повинен виглядати чистим, з відсутністю непотрібного синтаксису. Чистий код може навіть не відрізнятись від вбудованої особливості мови [13].

Подібно до інших функцій мови, Kotlin DSL є повністю статично типізованим. Це означає, що всі переваги статичної типізації, наприклад,

виявлення помилок компіляції та краща підтримка IDE залишається в силі, коли ви використовуєте моделі DSL для ваших API.

Яскравим прикладом застосування Kotlin DSL є такий програмний код, що повертає повну дату вчорашнього дня:

```
val yesterday = 1.days.ago
```

У противагу всім перевагам DSL має один недолік: може бути складно об'єднати їх з хост-додатком в загальноприйнятій мові. Вони мають власний синтаксис, який не можна безпосередньо вбудувати в програми на іншій мові програмування. Тому для виклику програми, написаної в DSL, потрібно або зберігати його в окремому файлі або вставляти його в рядок буквально. Це робить його нетривіальним для перевірки правильності взаємодії DSL з мовою хост-додатку під час компіляції, для відлагодження програми DSL, а також для надання коду IDE при його написанні. Крім того, окремий синтаксис вимагає окремого навчання і часто робить код важчим для читання [13]. Щоб вирішити цю проблему, зберігаючи більшість інших переваг DSL, ця концепція нещодавно отримала популярність внутрішніх DSL.

На відміну від зовнішніх DSL, які мають власний незалежний синтаксис, внутрішні DSL є частиною програм, написаних мовою загального призначення, використовуючи точно такий самий синтаксис. По суті, внутрішній DSL не є повністю окремою мовою, а скоріше конкретною спосіб використання мови загального призначення при збереженні ключових переваг DSL з незалежним синтаксисом.

Внутрішні конструкції Kotlin DSL називають «мовою» не просто так. Це відбувається завдяки тому, що DSL містить свою структуру, набір правил, який називається «граматикою». На природній мові, наприклад, українській, речення побудовані за допомогою слів, а також правил граматики за допомогою якої визначається, як ці слова можуть бути об'єднані один з одним. Аналогічно, у DSL, одиночна операція може складатися з декількох викликів функції, а перевірка типу гарантує, що ці виклики об'єднуються в правильному порядку [13].

В даній магістерській дисертації створення екземпляру сокета було реалізовано саме за допомогою внутрішніх конструкцій Kotlin DSL. Приклад програмного коду ініціалізації сокета за допомогою Kotlin DSL:

```
private val socket = createRxSocket {
    hostIp = "http://176.36.146.229"
    port = 9092
    namespace = "DHT22"
    options {
        forceNew = false
        reconnection = true
        reconnectionAttempts = 4
        reconnectionDelay = 1000
    }
}
```

Виклик методу createRxSocket повертає екземпляр класу RxSocket, за допомогою якого можна реєструвати події, на які будуть отримані дані від інших процесів, або ж відправляти власні дані до інших процесів за деякою визначеною подією. Поля для налаштування конфігурації сокету будуть детально описані в підрозділі 3.4.

Такий лаконічний запис ініціалізації сокету був створений за допомогою деяких визначених особливостей мови Kotlin:

1. Лямбда за межами дужок. Лямбда-вирази або лямбда – це блоки коду, які можна передавати у функції, зберігати або викликати їх. У мові Kotlin тип лямбда позначається наступним чином:

(список типів параметрів) -> тип, що повертається.

Слідуючи цьому правилу, найпримітивніший вид лямбда це () -> Unit, де Unit – це аналог Void в Java з одним винятком. В кінці лямбди або функції не потрібно писати конструкцію «*return* ...». Завдяки цьому завжди існує тип, що повертається, просто в Kotlin це відбувається неявно. У мові Kotlin діє наступне правило: якщо лямбда є останнім аргументом функції, то її можна винести за дужки, якщо при цьому лямбда єдиний параметр, то дужки можна не писати. В результаті, конструкція *x ({...})* може бути перетворена в *x () {}*, а потім, прибравши дужки, ми отримуємо *x {}*. При ініціалізації сокету було

використане це правило для того, щоб не писати багато разів надлишкові дужки, при цьому покращуючи читабельність програмного коду [13].

2. Лямбда з ресивером. Ідея доступу до полів зовнішнього типу без явного класифікатора може нагадувати про функції розширення, які дозволяють визначити власні методи для класів, визначених в іншому місці коду. Обидва розширення функції та лямбда з приймачами мають об'єкт приймача (ресивера), який повинен бути наданий коли функція викликається і доступна у своєму тілі. По суті, тип функції розширення описує блок коду, який може бути викликаний, як функція розширення [13]. На рис. 3.1 зображена структура лямбди з ресивером, у якому існує ресивер типу String, два параметри типу Int, що повертають результат типу Unit.

Рисунок 3.1 – Зображення структури лямбди з ресивером

Переважна більшість коду, що реалізовує DSL ініціалізації сокетів були побудовані за допомогою лямбд з ресивером. Реалізований метод створення екземпляру сокета у програмній бібліотеці обміну даними між процесами виглядає так:

```
fun createRxSocket(block: RxSocketBuilder.() -> Unit =
   RxSocketBuilder().apply(block).build()
```

При цьому змінна block являється лямбдою з ресивером, в якому ресивером виступає клас RxSocketBuilder, що відповідає за встановлення конфігурації. Метод build() повертає екземпляр класу RxSocket із встановленими полями конфігурації сокету:

```
fun build() = RxSocket(hostIp, port, namespace, options, gson,
socketLoggingInterceptor)
```

В класі RxSocketBuilder існують поля для задання як основної конфігурації, так і опціональної, яка визначається об’єктом Options. Створення такого об’єкту практично не відрізняється від створення сокету:

```
fun options(block: OptionsBuilder.() -> Unit) {
    options = OptionsBuilder().apply(block).build()
}
```

В класі OptionsBuilder створюється екземпляр класу Options, в якому визначені всі опціональні конфігурації сокета:

```
fun build(): Options {
    val options = Options()

    /*                          ...                          */
    /* Встановлення всіх визначених полів класу Options */
    /*                          ...                          */

    return options
}
```

Отже, завдяки предметно-орієнтованим конструкціям мови Kotlin була створена DSL, за допомогою якої можливе створення екземпляру сокета на вищому рівні абстракції, ніж при програмуванні в імперативному стилі.

3.2. Події, що реєструються за допомогою програмної бібліотеки

Події, що реєструються за допомогою програмної бібліотеки обміну даними між процесами, реалізованої в даній магістерській дисертації, умовно можна поділити на дві категорії:

– події, пов’язані зі станом сокету;

– події, визначені розробником для реалізації бізнес логіки.

Для того, щоб відслідковувати події, визначені розробником створені такі методи:

1. **observableOn**. Метод містить агрумент eventName, що визначає назву події, яка буде прослуховуватись. Також, метод містить аргумент returnClass, що визначає клас, який має бути отриманий від сервера. Дані, що відправляються від клієнта до сервера, та навпаки, існують у формі JSON, тобто звичайного рядка. Програмна бібліотека, розроблена в даній

магістерській дисертації підтримує можливість автоматичної десеріалізації рядку у форматі JSON в об’єкт такого класу, який визначений в аргументі returnClass. Реалізація методу observableOn наведена нижче:

```
fun <T : Any> observableOn(eventName: String, returnClass:
Class<T>): Observable<T> {
    checkSubscribedToEvent(eventName)
    return Observable.create<T> { emitter ->
        val listener = getEmitterListener<T>(emitter,
eventName, returnClass)
        socket.on(eventName, listener)
        socketEvents.add(eventName)
    }.doOnSubscribe {
        if (!it.isDisposed) {
            compositeDisposable.add(it)
        }
    }
}
```

Метод checkSubscribeToEvent(eventName) перевіряє чи була вже зареєстрована подія з такою самою назвою раніше. Якщо подія не була зареєстрована, тоді відбудеться реєстрація, в іншому випадку виникне помилка EventAlreadySubscribedException, що виведе розробнику текст про те, що подія з такою назвою вже зареєстрована. Можна було б обійтись без цієї перевірки, та не створювати можливості виникнення критичної помилки. Проте, було вирішено додати її, оскільки вона може виникнути лише на етапі розробки, і якщо буде перевизначена подія з однаковим ім’ям, це може створити випадок, коли розробник очікуватиме іншого результату і не розумітиме чому подія опрацьовується не коректно. Реалізація методу наведена нижче:

```
private fun checkSubscribedToEvent(event: String) {
    if (socketEvents.contains(event)) {
        throw EventAlreadySubscribedException(event)
    }
}
```

Метод observableOn повертає джерело даних типу Observable, для цього створюється клас Emitter.Listerner для прослуховування події. Створення такого класу відбувається за допомогою методу getEmitterListener. Це метод, в якому відбувається десеріалізація даних, що приходять від сервера в об’єкт

класу, що вказаний в аргументі returnClass. Десеріалізація в програмній бібліотеці є гнучкою, оскільки десеріалізатор може бути встановлений розробником. Варто зауважити, що встановлений десеріалізатор за замовчуванням є GSON. Перед десеріалізацією вхідний об'єкт перевіряється на наявність і, якщо його немає, тоді розробник отримає помилку EmptySocketDataException. Також, вхідний об'єкт перевіряється на можливість десеріалізації і, якщо це неможливо, тоді розробник отримає помилку EventJsonSyntaxException. Реалізацію методу getEmitterListener наведено нижче:

```
private fun <T : Any> getEmitterListener(
  emitter: io.reactivex.Emitter<T>,
  eventName: String, returnClass: Class<T>) =

    Emitter.Listener { args ->
      if (args == null || args[0] == null) {
emitter.onError(EmptySocketDataException(eventName))
      } else {
          try {
            val data = gson.fromJson<T>(
               args[0].toString(), returnClass)
            emitter.onNext(data)
          } catch (e: JsonSyntaxException) {
            emitter.onError(EventJsonSyntaxException(
               eventName, e.message))
          }
      }
    }
```

Після отримання екземпляру класу Emitter.Listener відбувається його реєстрація відносно назви події. Назва події та її підписка додається в список подій для того, щоб можна було коректно закрити з'єднання та вивільнити всі підписки на події з пам'яті.

2. flowableOn. Цей метод є ідентичним методу observableOn, як і його реалізаця, зокрема того, що він повертає не Observable, як джерело даних, а Flowable. В такому випадку таке джерело матиме змогу опрацьовувати Backpressure стан, а саме такий стан, при якому генерація даних джерелом є швидшою ніж його обробка споживачем. Через аргумент backpressureStrategy передається режим обробки стану Backpressure. За замовчуванням, вказана

стратегія DROP, при якій відбуватиметься відкидання тих елементів, які неможливо обробити.

Для того, щоб відслідковувати події, пов’язані зі станом сокету, були створені такі методи:

– observableOnConnect() та flowableOnConnect(). Джерела даних, що надають повідомлення про успішне встановлення з’єднання сокету;

– observableOnConnecting() та flowableOnConnecting(). Джерела даних, що надають повідомлення про те, що на даний момент відбувається встановлення з’єднання;

– observableOnConnectError() та flowableOnConnectError(). Джерела даних, що надають повідомлення про помилку встановлення з’єднання. Дані джерела надають не тільки сповіщення, а й текст помилки з’єднання сокетів;

– observableOnConnectTimeout() та flowableOnConnectTimeout(). Джерела даних, що надають повідомлення про помилку з’єднання через те, що вийшов час встановлення, визначений в конфігурації;

– observableOnDisconnect() та flowableOnDisconnect(). Джерела даних, що надають повідомлення про успішний розрив з’єднання сокетів;

– observableOnError() та flowableOnError(). Джерела даних, що надають повідомлення про помилку, що виникла після успішного встановлення з’єднання. Дані джерела надають не тільки сповіщення, а й текст помилки;

– observableOnMessage() та flowableOnMessage(). Джерела даних, що надають можливість видачі повідомлень від сервера до клієнта та навпаки;

– observableOnPing() та flowableOnPing(). Джерела даних, що надають повідомлення про успішну відправку пакету «ping» на сервер;

– observableOnPong() та flowableOnPong(). Джерела даних, що надають повідомлення про отримання пакету «pong» від сервера;

– observableOnReconnect() та flowableOnReconnect(). Джерела даних, що надають повідомлення про успішне повторне встановлення з’єднання сокета;

– observableOnReconnecting() та flowableOnReconnecting(). Джерела даних, що надають повідомлення про те, що на даний момент відбувається повторне встановлення з'єднання. Джерела надають не тільки сповіщення, а й номер спроби повторного підключення в полі attempt;

– observableOnReconnectAttempt() та flowableOnReconnectAttempt(). Джерела даних, що надають повідомлення про спробу повторного з'єднання сокетів;

– observableOnReconnectError() та flowableOnReconnectError(). Джерела даних, що надають повідомлення про помилку повторного з'єднання сокетів. Дані джерела надають не тільки сповіщення, а й текст помилки;

– observableOnReconnectFailed() та flowableOnReconnectFailed(). Джерела даних, що надають повідомлення про помилку повторного з'єднання сокетів за задану кількість спроб, що визначена в конфігурації. Після отримання сповіщення від цих джерел повторне підключення сокетів припиняється;

– observableOnSendDataError() та flowableOnSendDataError(). Джерела даних, що надають повідомлення про помилку при відправлені даних до сокету, оскільки з'єднання розірване.

– observableOnGenericEvent() та flowableOnGenericEvent(). Джерела даних, що надають повідомлення про те, яка з вищеперелічених подій була викликана. Для цього ці джерела надають об'єкт перелічуваного типу RxSocketEvent. Його реалізація виглядає так:

```
enum class RxSocketEvent {
    CONNECTED,
    CONNECTING,
    CONNECT_ERROR,
    CONNECT_TIMEOUT,
    DISCONNECTED,
    ERROR,
    MESSAGE, PING,
    PONG, RECONNECTED,
    RECONNECTING, RECONNECT_ATTEMPT,
    RECONNECT_ERROR, RECONNECT_FAILED,
    SEND_DATA_ERROR
}
```

Варто зауважити, що немає сенсу реєструвати кожну подію по окремості, якщо варто забезпечувати прослуховування всіх подій. Для цього і були створені ці два джерела. Їх реалізацією є просте злиття вищеописаних джерел в один за допомогою методу merge(). Демонстрація реалізації на прикладі методу observableOnGenericEvent() зображено в коді нижче:

```
fun observableOnGenericEvent() =
    Observable.merge(listOf(
        observableOnConnect().map { CONNECTED },
        observableOnConnecting().map { CONNECTING },
        observableOnConnectError().map { CONNECT_ERROR },
        observableOnConnectTimeout().map { CONNECT_TIMEOUT },
        observableOnDisconnect().map { DISCONNECTED },
        observableOnError().map { ERROR },
        observableOnMessage().map { MESSAGE },
        observableOnPing().map { PING },
        observableOnPong().map { PONG },
        observableOnReconnect().map { RECONNECTED },
        observableOnReconnecting().map { RECONNECTING },
        observableOnReconnectAttempt().map {RECONNECT_ATTEMPT},
        observableOnReconnectError().map { RECONNECT_ERROR },
        observableOnReconnectFailed().map { RECONNECT_FAILED },
        observableOnSendDataError()?.map { SEND_DATA_ERROR }))
```

Відмінність між кожною парою подій лише у наявності підтримки backpressure, оскільки observableOn… повертає джерело даних Observable, а flowableOn… повертає джерело даних Flowable. Варто зауважити, що за замовчуванням виставлена стратегія BackpressureStrategy.DROP, при якій відбуватиметься відкидання тих елементів, які неможливо обробити.

За допомогою вищеописаних системних подій, розробник може показувати на пристрої стан з’єднання, що може бути інформативним для користувача. Варто зауважити, що реєстрація системних подій не є обов’язковою. Ці події призначені лише для інформативних цілей.

3.3. Поля для налаштування конфігурації сокету.

Для того, щоб відправка даних відбувалась коректному адресату необхідно правильно налаштувати конфігурацію екземпляру сокету. Для цього був створений клас RxSocketBuilder, в якому можна прописати всі налаштування засобами DSL конструкцій.

Серед полів налаштування конфігурації варто відзначити наступні:

1. hostIp – кінцева IP-адреса сервера, задається у форматі: «http:/ domain_name» або «https://domain_name», де domain_name – назва домену, або визначена IP адреса. За замовчуванням кінцева IP адреса не встановлена. Для того, щоб попередити розробника про те, що необхідно її встановити буде відображено помилку «PLEASE SET HOST IP AND PORT»;

2. port – відкритий порт, на якому прослуховує дані сервер, задається у форматі числа. Кількість портів обмежено з врахуванням 16-бітної адреси (2^{16} = 65536, початок - «0»). Всі порти розділені на три діапазони – загальновідомі (або системні, 0-1023), зареєстровані (або користувацькі, 1024-49151) і динамічні (або приватні, 49152-65535). За замовчуванням встановлений порт 0;

3. namespace – простір імен, на якому відбуватиметься прослуховування подій сокета. Задається у форматі строки. За замовчуванням простору імен не встановлено. Таке поле є необов'язковим, проте для коректної роботи рекомендується визначення даного поля як на сервері, так і на клієнті в залежності від вимог бізнес логіки;

4. serializer – встановлення екземпляру визначеного серіалізатора. Такий серіалізатор використовуватиметься для того, щоб коректно розпізнати дані, які приходять від сервера. За замовчуванням виставлено серіалізатор GSON;

5. socketLoggingInterceptor – встановлення екземпляру визначеного логера. Такий логер використовуватиметься для логування системних подій, подій визначених користувачем, а також даних, що відправляються від клієнта до сервера;

6. options – екземпляр класу Options, який включає в себе такі поля:

6.1. forceNew – поле булевого типу. За замовчуванням встановлене значення false. За замовчуванням при розриві з'єднання буде кешуватись і, якщо з'єднання буде відновлене, тоді використовуватиметься вже створений екземпляр з'єднання. Якщо не потрібно повторно використовувати екземпляр кешованого сокета, коли параметр запиту змінюється, тоді слід скористатися

полем forceNew. Наприклад: якщо додаток дозволяє користувачеві вийти з існуючого аккаунту, а новому користувачу зайти в свій приватний. За допомогою такого механізму забезпечується інкапсуляція даних в екземплярі сокета.

6.2. reconnection – поле булевого типу, яке визначає чи слід автоматично перепідключатись до сервера в разі виникнення помилки з’єднання. За замовчування встановлене значення true;

6.3. reconnectionAttempts – поле булевого типу, яке визначає кількість спроб перепідключення до сервера в разі виникнення помилки з’єднання. За замовчуванням визначене значення 0, що означає нескінченну кількість спроб;

6.4. reconnectionDelay – поле типу Long, яке визначає як довго необхідно чекати, перш ніж спробувати нове підключення. Вимірюється в мілісекундах. Таке значення буде визначене в залежності від поля randomizationFactor. За замовчуванням визначене значення 1000 мс;

6.5. randomizationFactor – поле типу Double, яке повинно бути встановлене в межах від 0 до 1. Воно впливає на поле reconnectionDelay та reconnectionDelayMax. За замовчуванням встановлене значення 0.5. Наприклад: якщо reconnectionDelay дорівнює 1000, а randomizationFactor дорівнює 0.5, тоді початкова затримка буде визначена від 500 мс до 1500 мс;

6.6. reconnectionDelayMax – поле типу Long, що визначає максимальний час очікування між підключенням. За замовчуванням виставлене значення 5000. Вимірюється в мілісекундах. Кожна спроба збільшує затримку повторного з'єднання на 2 рази разом з рандомізацією, як описано в прикладі вище;

6.7. timeout – поле типу Long, що визначає тайм-аут з’єднання перед викликом подій connect_error і connect_timeout. Вимірюється в мілісекундах. Значення за замовчуванням 20000 мс.

3.4. Логування подій

Відображення стану програми є дуже важливим не тільки на етапі розробки програмного продукту, а й при його використанні. Аналіз додатку на кожному його кроці дозволяє зменшити кількість явних і неявних помилок, а також запобігти його неочікуваних фатальних завершень. Цього всього можна досягти завдяки логуванню.

Логування – це створення текстових записів про виконання визначених подій протягом виконання програми. Логування може відбуватись у файл, базу даних, на сервер за допомогою API, тощо. Після чого логи можуть бути використані для аналізу виконання бізнес логіки.

Нині існує досить багато програмних бібліотек для логування подій, таких як Timber, Log4J, SLF4J, тощо. Кожен з них має свої переваги та недоліки, проте основна ідея перелічених бібліотек однакова – запис інформації про виконання програмного коду.

Оскільки програмна бібліотека розроблена в даній магістерській дисертації працює переважно з подіями та даними, тому система логування неймовірно необхідна. Для цього була впроваджена інтеграція за допомогою класу SocketLoggingInterceptor. Це є звичайний інтерфейс, який описує два методи: logInfo(message: String) та logError(message: String). Сам інтерфейс виглядає так:

```
interface SocketLoggingInterceptor {
    fun logInfo(message: String)
    fun logError(message: String)
}
```

Метод logInfo(message) створює лог-повідомлення інформаційного тексту, а метод logError(message) створює лог-повідомлення тексту про помилку.

Для того, щоб додати можливість логування подій та відправлених даних в даній програмній бібліотеці необхідно лише додати до конфігурації об’єкт, що реалізує інтерфейс SocketLoggingInterceptor в конфігурацію при створенні об’єкту RxSocket так, як показано в наступному фрагменті коду:

```
fun provideSocket(gsonProvided: Gson, loggingInterceptor:
SocketLoggingInterceptor): RxSocket {
    return createRxSocket {
        hostIp = "http://176.36.81.205"
        port = 9092
        namespace = "raspberry"
        socketLoggingInterceptor = loggingInterceptor
        gson = gsonProvided
        options {
            forceNew = false
            reconnection = true
        }
    }
}
```

Впровадження класу, що реалізує інтерфейс SocketLoggingInterceptor виглядає так, як показано в коді нижче:

```
fun provideSocketLoggingInterceptor(someLogger: SomeLogger):
SocketLoggingInterceptor {
    return object : SocketLoggingInterceptor {
        override fun logInfo(message: String) {
            someLogger.i(message)
        }

        override fun logError(message: String) {
            someLogger.e(message)
        }
    }
}
```

Отже, в розділі 3 було наведено структуру та алгоритм роботи програмної бібліотеки обміну даними між процесами, розробленої в даній магістерській дисертації. Був описаний механізм створення екземпляру сокета, що забезпечується за допомогою розроблених Kotlin DSL конструкцій. Для отримання даних з сервера після створення з'єднання сокету необхідно зареєструвати події, та опрацювати дані, що отримаються через сокет засобами реактивного програмування. Слід не забувати про можливість логування подій та даних, що передаються через сокет, оскільки аналіз таких інформаційних повідомлень може забезпечити покращення комерційної привабливості програмного продукту.

4. МЕТОДИКА ВИКОРИСТАННЯ ПРОГРАМНОЇ БІБЛІОТЕКИ

Для використання програмної бібліотеки обміну даними між процесами варто користуватись наступним визначеним алгоритмом:

1. Створення екземпляру сокету.
2. Задання конфігурації, а саме: кінцевої IP адреси (або домену) серверу, порту, забезпечення можливості кешування з'єднання, встановлення транспортного протоколу, забезпечення можливості повторного з'єднання, встановлення кількості спроб повторного з'єднання, тощо.
3. Реєстрація подій, пов'язаних зі станом сокету, а саме: створення, розриву, помилки створення, повтору створення з'єднання, тощо.
4. Реєстрація подій, визначених розробником.
5. Відправлення даних через сокет.
6. Закриття з'єднання та видалення сокету.

4.1. Створення екземпляру сокету

Для створення екземпляру сокета використовується описана в розділі 3.1 внутрішня предметно-орієнтована мова Kotlin DSL. Приклад створення екземпляру сокету зображено на наступному програмному коді:

```
private val socket = createRxSocket {
    hostIp = "http://176.36.146.229"
    port = 9092
    gson = Gson()
    namespace = "DHT22"
    socketLoggingInterceptor = loggingInterceptor
    options {
        forceNew = false
        reconnection = true
        reconnectionAttempts = 3
        reconnectionDelay = 1000
        reconnectionDelayMax = 10000
        randomizationFactor = 0.3
        timeout = 5000
    }
}
```

Після виконання вищеописаного програмного коду в змінній socket буде міститись посилання на екземпляр класу RxSocket, який буде відправляти та отримувати дані від сервера з кінцевою адресою 176.36.146.229, буде містити незахищене з'єднання (без протоколу SSL), на порті 9092. Для серіалізації даних, що отримуватимуться з сервера, буде використовуватись серіалізатор GSON. Для логування подій використовуватиметься об'єкт, що реалізовує інтерфейс SocketLoggerInterceptor. Всі дані будуть проходити за адресою http://176.36.146.229/DHT22, оскільки DHT22 було визначене як простір імен для даного екземпляру сокета. При розриві з'єднання буде відбуватись автоматичне перепідключення 3 рази поспіль з інтервалом від 700 мс до 1300 мс. Час між подіями connect_error і connect_timeout дорівнює 5 секунд.

4.2. Відправка даних

Для відправки даних до іншого процесу використовуються два однакових методи з різними аргументами:

1. Метод для передачі будь-яких об'єктів за визначеною подією:

```
fun <T> sendData(eventName: String, vararg data: T) {
    if (socket.connected()) {
        socketLoggingInterceptor?.logInfo("RxSocket. Custom
            event $eventName. Send data:
            ${Arrays.toString(data)}")
        socket.emit(eventName, data);
    } else {
        socketLoggingInterceptor?.logError("RxSocket. Custom
            event $eventName. Error while sending data:
            ${Arrays.toString(data)}. Socket is
            disconnected.")
        systemSubjects[SEND_DATA_ERROR]!!.onNext(Unit)
    }
}
```

Аргумент eventName визначає ім'я події, за допомогою якої відбудеться передача даних. Аргумент data є змінною перечислювального типу, адже через сокет можна відправляти не лише один елемент, а й список таких елементів. Варто зауважити, що передача даних відбувається лише по відкритому з'єднанню, тому при передачі даних з закритим з'єднанням відбудеться відправка події SEND_DATA_ERROR. В такому випадку розробник зможе

правильно відобразити користувачу помилку при відправленні даних з закритим з'єднанням. Важливо відмітити, що метод sendData не повертає ніяких даних, а отже має тип Unit.

Наприклад: для передачі рядка «NTUU KPI» по назві події «Diploma» необхідно просто викликати даний метод з визначеними аргументами, попередньо ініціалізувавши екземпляр сокета:

```
socket.sendData("Diploma", "NTUU KPI")
```

В такому випадку сервер успішно отримає рядок, при наявності з'єднання з клієнтом.

2. Метод для передачі будь-яких об'єктів за визначеною подією з підтвердженням.

```
fun <T> sendData(eventName: String,
                 vararg data: T,
                 acknowledgment:(args: Array<out Any>)->Unit) {
    if (socket.connected()) {
        socketLoggingInterceptor?.logInfo("RxSocket. Custom
           event $eventName. Send data:
           ${Arrays.toString(data)}")

        socket.emit(eventName, data) { acknowledgment(it) };
    } else {
        socketLoggingInterceptor?.logError("RxSocket. Custom
           event $eventName. Error while sending data:
           ${Arrays.toString(data)}. Socket is
           disconnected.")

        systemSubjects[SEND_DATA_ERROR]!!.onNext(Unit)
    }
}
```

Аргументи eventName та data виконують ті ж функції, які описані в попередньому пункті. Аргумент acknowledgement є змінною типу функція, що містить внутрішній аргумент args та повертає результат типу Unit, тобто є звичайним методом. В змінній args містяться об'єкти, що будуть отримані від сервера при отриманні підтвердження. Як і у попередньому випадку, в даному методі відбувається перевірка на наявність коректного з'єднання між сокетами.

Наприклад: для передачі рядка «NTUU KPI» по назві події «Diploma» та для отримання підтвердження «Gotcha» необхідно викликати даний метод з визначеними аргументами, попередньо ініціалізувавши екземпляр сокета:

```
socket.sendData("Diploma", "NTUU KPI") {
    if(it.isNotEmpty()){
       Log.d("TAG", "Acknowledgement ${it as String}")
    }
}
```

4.3. Реєстрація подій

Події в програмній бібліотеці розділяються на системні та такі, що визначаються розробником при реалізації конкретної бізнес логіки.

Для реєстрації подій, що визначаються розробником використовується два методи:

- observableOn();
- flowableOn().

Для реєстрації події без підтримки backpressure необхідно створити джерело даних Observable, за допомогою методу observableOn та підписатись на нього. Далі все залежить від бізнес логіки, яку реалізовує розробник. Приклад використання методу observableOn наведено нижче:

```
socket.observableOn("loginResult", HashMap::class.java)
        .subscribe {
            Log.d("TAG", "Login result: $it")
        }
```

При підписці на джерело даних, метод subscribe повертає екземпляр класу Disposable. Такий об’єкт використовується для того, щоб відписатись від джерела даних після завершення роботи з сокетом. Це означає, що необхідно створювати список таких об’єктів при реєстрації кожної події, та вивільнювати їх. Це було реалізовано в програмній бібліотеці, тому розробнику, що користуватиметься нею не потрібно буде обробляти цю ситуацію. Необхідно буде лише виконати метод close() після завершення роботи. Варто звернути увагу на те, що IDE підсвічуватиме вищеописаний код з попередженням «результат методу subscribe не використовується». З впевненістю можна сказати, що таке попередження можна ігнорувати і додати

аннотацію @SuppressLint("CheckResult") над методами використання для того, щоб код не підсвічувався жовтим кольором.

Для реєстрації події з підтримкою backpressure необхідно створити джерело даних Flowable, за допомогою методу flowableOn та підписатись на нього. Далі все залежить від бізнес логіки, яку реалізовує розробник. Приклад використання методу flowableOn наведено нижче:

```
socket.flowableOn("loginResult", HashMap::class.java,
BackpressureStrategy.BUFFER)
        .subscribe {
            Log.d("TAG", "Login result: $it")
        }
```

На відміну від створення Observable, в методі flowableOn існує аргумент backpressureStrategy, що визначає стратегію опрацювання тих елементів, які клієнт не в змозі опрацювати. В вищеописаному прикладі такі елементи будуть додані в буфер, та опрацьовані тоді, коли клієнт зможе їх опрацювати.

Для реєстрації системних подій необхідно обрати: варто опрацьовувати стан Backpressure чи ні. Дуже малоймовірно, що виникне такий стан, коли клієнт не встигатиме опрацьовувати події, пов’язані з станом сокету, оскільки вони приходять не так швидко. Проте, існують такі пристрої, на яких це варто робити.

Якщо при ініціалізації сокету була задана конфігурація логера, тоді рекомендується реєструвати кожну подію окремо. Немає необхідності логувати кожну подію при тривіальній логіці програми. Приклад реєстрації системної події зображено в наступному фрагменті програмному коді на прикладі методу observableOnConnect():

```
rxSocket.observableOnConnect()
        .subscribe(
                { Log.i("TAG", "Socket connected") },
                { Log.e("TAG","An error occurred when socket"+
                        " connected event emitted.") })
```

В першій лямбда-функції виконається код при умові успішного отримання події про з’єднання. В другій лямбда-функції виконається програмний код при умові помилки отримання цієї події.

Якщо розробнику необхідно реєструвати всі системні події сокета одночасно, тоді варто скористуватись методом observableOnGenericEvent(). В такому випадку буде приходити екземпляр класу перелічувального типу RxSocketEvent. Приклад використання даного методу зображено в наступному фрагменті програмного коду:

```
rxSocket.observableOnGenericEvent()
        .subscribe {
            when(it){
                CONNECTED -> Log.i("TAG", "Connected")
                DISCONNECTED -> Log.i("TAG", "Disconnected")
                RECONNECTING -> Log.i("TAG", "Reconnecting")
                RECONNECTED -> Log.i("TAG", "Reconnected")
                PING -> Log.i("TAG", "Ping")
                PONG -> Log.i("TAG", "Pong")
                ERROR, RECONNECT_ERROR, CONNECT_ERROR ->
                    Log.i("TAG", "Error")
            }
        }
```

В такому випадку буде відбуватись логування всіх подій, навіть тих, що не перелічені вище. Проте, якщо логер не зареєстрований, або ж надмірна кількість логів рахується нормальною, тоді в такому випадку це найкращий спосіб прослідковування системного стану сокета.

4.4. Закриття з'єднання та видалення екземпляру сокету

Після передачі та отриманні даних через сокет необхідно пам’ятати про закриття з’єднання та вивільнення оперативної пам’яті, оскільки можуть виникнути витоки пам’яті. Для цього клас RxSocket реалізує інтерфейс Closeable. Це означає, що в класі наявний метод close(), який необхідно викликати в тому випадку, коли настав час закрити з’єднання сокету та завершити роботу з ним.

Метод close не лише закриває з’єднання сокета, а й збуває всі підписки, які були створені для прослуховування як подій пов’язаних зі станом сокета, так і подій визначених розробником. Також, метод відключається від прослуховування всіх подій, що були відкриті. Реалізація методу виглядає так:

```
override fun close() {
    if (socket.connected()) {
        socket.disconnect()
    }

    if (!compositeDisposable.isDisposed) {
        compositeDisposable.dispose()
    }
    compositeDisposable = CompositeDisposable()

    for (subscription in compositeSubscription) {
        subscription.cancel()
    }
    compositeSubscription.clear()

    for (event in socketEvents) {
        socket.off(event)
    }
}
```

Оскільки клас RxSocket реалізує інтерфейс Closeable, було створену можливість автоматичного відкривання та закривання з'єднання за допомогою розширювальної функції use, її реалізація виглядає так:

```
fun RxSocket.use(block: RxSocket.() -> Unit) {
    this.connect()
    apply(block)
    this.close()
}
```

За допомогою цієї функції можна відправляти дані і при цьому не турбуватись про відкриття та закриття з'єднання, наприклад:

```
 socket.use {
    sendData("Diploma", "NTUU KPI")
    sendData("Name", "Andrii Chernysh")
    sendData("Faculty", "Applied Mathematics")
}
```

Варто відзначити, що при використанні цієї функції не потрібно додавати методи прослуховування подій, оскільки вони не працюватимуть, так як реєстрацію подій необхідно виконувати до виконання методу RxSocket.connect(). Така конструкція створена лише для швидкої відправки даних. Якщо необхідно отримати дані від сервера, тоді можна використовувати метод sendData з аргументом acknowledgement, в такому випадку можна буде отримати дані про підтвердження від сервера.

5. ПРИКЛАД ВИКОРИСТАННЯ ПРОГРАМНОЇ БІБЛІОТЕКИ ОБМІНУ ДАНИМИ МІЖ ПРОЦЕСАМИ

Часто трапляється так, що дії людини призводять не завжди до правильних рішень. Яскравим прикладом такого є трагічна подія на Чорнобильській атомній електростанції, коли персонал відключив ряд технічних засобів захисту і порушив важливі положення регламенту експлуатації в частині безпечного ведення технологічного процесу. Щодня людство розробляє нові технології, які здатні автоматизувати виробничий. Завдяки інтернету люди можуть обмінюватись інформацією на різні відстані з різною швидкістю. Якщо ж отримувати сигнали від зовнішніх пристроїв, обробляти і передавати їх за допомогою інтернету – ми отримаємо такий термін як «Інтернет речей».

Термін «Інтернет речей» вперше було сформульовано в кінці XX-го століття, у 1999 році. Це концепція комунікації об’єктів (речей), які використовують технології для взаємодії між собою та з навколишнім середовищем. Також ця концепція передбачає виконання пристроями певних дій без втручання людини.

Приклад використання програмної бібліотеки обміну даними між процесами відноситься до даної категорії. Для демонстрації роботи програмної бібліотеки була створена програмно-апаратна модель на базі мікроконтролера Raspberry Pi 3, датчика температури та вологості DHT22, світлодіода та смартфона з операційною системою Android.

Для демонстрації роботи програмної бібліотеки обміну даними між процесами існуючі рішення пропонують такі способи, як Web-сервер, мобільні пристрої (смартфони), чат-боти, смарт годинники, тощо. У даній магістерській дисертації для прикладу використовується смартфон з операційною системою Android.

За даними інформаційного порталу «StatCounter» платформа Android займає високу долю ринку мобільних пристроїв, що продемонстровано на рис. 5.1.

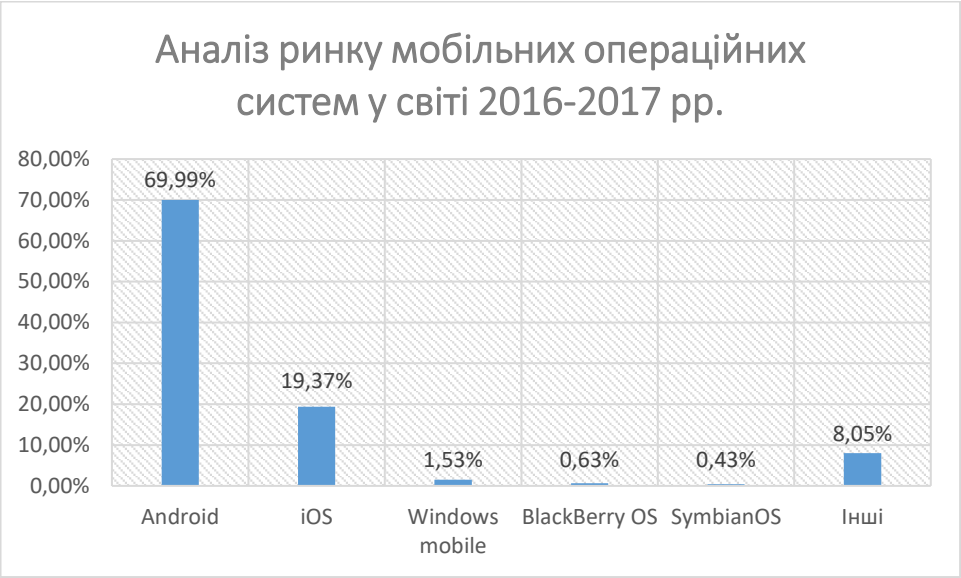

Рисунок 5.1 - Діаграма аналізу ринку мобільних операційних систем у світі за 2016-2017 роки.

Дослідження, проведене Київським Міжнародним Інститутом Соціології на замовлення агентства мобільного маркетингу LEAD9, показало, що 41% людей у віці 18-50 років користуються сенсорними смартфонами. А серед молодих людей у віці 18-30 ця цифра становить вже 59% (діаграма наведена на рис. 5.2). Серед жителів окремих регіонів ситуація, в цілому, дуже схожа [14].

Саме через високу популярність і надійність мобільних смартфонів з операційною системою Android їх було обрано як «клієнт», адже смартфон є у кожного четвертого Українця.

Кінцевим результатом є – асинхронне отримання даних про температуру та вологість на смартфон з датчика та керування світлодіодом. Асинхронний обмін даними між процесами забезпечується за допомогою програмної бібліотеки обміну даними між процесами, що описана у даній магістерській дисертації.

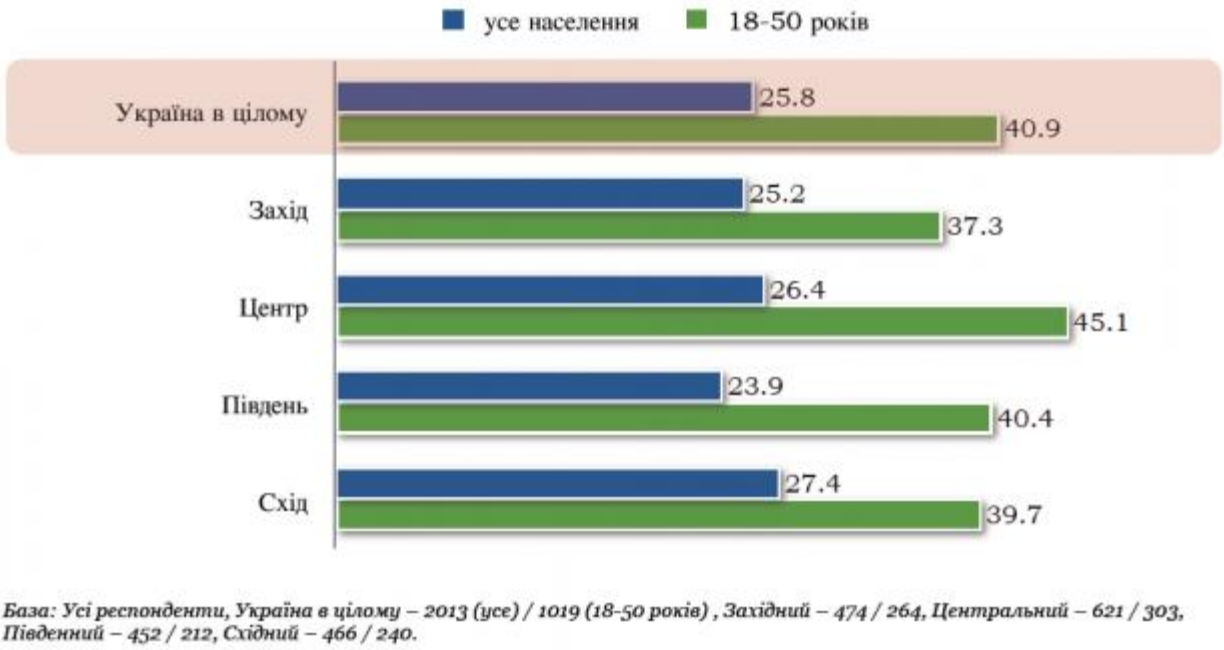

Рисунок 5.2 - Діаграма порівняння кількості користувачів смартфонів в залежності від віку та регіону України.

5.1. Мікроконтролер Raspberry Pi 3

Для реалізації прикладу використання програмної бібліотеки обміну даними між процесами було обрано одноплатний комп’ютер Raspberry Pi 3, розроблений британським фондом Raspberry Pi Foundation.

Raspberry Pi побудований на системі-на-чипі (SoC) Broadcom BCM2835, яка включає в себе процесор ARM із тактовою частотою 700 МГц, графічний процесор VideoCore IV, і 512 чи 256 мегабайт оперативної пам'яті. Твердий диск відсутній, натомість використовується SD карта [15]. Схематично RPi3 зображено на рис. 5.3.

Комп'ютер планувався як пристрій для навчання дітей програмуванню, однак здобув популярність і в інших сферах — зокрема, на його основі роблять проект «розумний дім». Найдешевий Raspberry Pi поставляється без корпусу і має вигляд плати розміром з кредитну карту. Плата важить 45 грамів. У комп'ютері задіяний 700-мегагерцевий процесор на архітектурі ARM; присутній роз'єм для навушників і слот для карти пам'яті. Молодша (A) і старша (B) моделі Raspberry Pi відрізняються об'ємом оперативної пам'яті (256 мегабайт проти 512 мегабайт) і кількістю USB-портів (один проти двох). Крім

цього, у старшої моделі є роз'єм Ethernet 10/100, а молодша споживає на третину менше енергії [15].

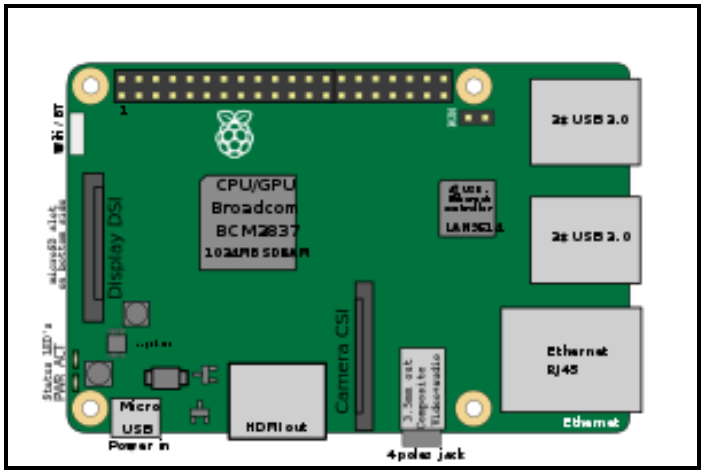

Рисунок 5.3 – Схематичне зображення одноплатного комп’ютера Raspberry Pi 3.

Raspberry Pi 3 підтримується стандартним набором операційних систем, в тому числі Raspbian (офіційний варіант Debian), а також Debian Wheezy, Ubuntu Mate, Fedora Remix. У Raspbian сьогодні вбудована маса додатків для навчання і програмування на Python (основною мовою роботи з Raspberry), безкоштовна версія Wolfram Mathematica. Фірмовий дистрибутив для Windows 10 працює на платі третьої версії все так же, через PowerShell, з підтримкою тільки 32-бітних додатків. Всі ці ОС давно знайомі ентузіастам і відмінно оновлюються.

Чіпи ЦП першого і другого покоління плати Raspberry Pi не вимагає охолодження, якщо чіп не був розігнаний, але Raspberry Pi 2 SoC може нагріватися більше, ніж зазвичай при розгоні [15].

Більшість чіпів Raspberry Pi можна розігнати до 800 МГц, а деякі до 1000 МГц. Існують відгуки, що Raspberry Pi 2 може бути так само розігнаний, в крайніх випадках, навіть до 1500 МГц (відкидаючи всі функції безпеки і обмеження перенапруження). У дистрибутиві Raspbian Linux опцію розгону при завантаженні можна зробити за допомогою команди програмного забезпечення «sudo raspi-config» без анулювання гарантії. У таких випадках

Raspberry Pi автоматично відміняє розгін, якщо чіп досягає 85 °C, проте можна перевизначити автоматичні налаштування перенапруження і розгону. Для таких випадків потрібно встановити радіатор належного розміру, щоб захистити чіп від серйозного перегріву [15].

Нові версії прошивок містять можливість вибору між п'ятьма режимами розгону, які при використовуються для збільшення продуктивності SoC без шкоди для терміну служби плати. Це робиться шляхом моніторингу температури ядра чіпа, завантаження процесора і динамічного регулювання тактової частоти і напруги ядра.

Однією з найцікавіших особливостей Raspberry Pi є наявність портів GPIO (general purpose input / output). Завдяки цьому комп'ютер можна використовувати для управління різними периферійними пристроями. У моделі «B» плати присутній 26-піновий, а в моделі «B+» і «2 B» - 40-піновий роз'єм GPIO [15].

Комп'ютер поширюється повністю зібраними на чотиришаровій друкованій платі розміром з банківську карту. У стандартний комплект поставки входить тільки сама плата. Корпус, блок живлення, флеш-карту необхідно замовляти окремо. Щодо блоку живлення, так як RPi3 живиться напругою 5В, згодиться звичайний USB порт від комп’ютера, ноутбука і навіть телевізора або блок живлення сучасних смартфонів. Вхід для живлення одноплатного комп’ютера – microUSB. Проте потрібно зауважити, що блок живлення має видавати силу струму 2.5 А. Максимальний струм виходу кожного піна не повинен перевищувати 16мА. Сумарний струм виходу всіх пінів не повинен перевищувати 50мА. 5-ти вольтові піни можуть давати високий струм, який залишається після живлення RPi3 та інших периферійних пристроїв – до 500мА.

5.2. Датчик температури та вологості DHT22

Отримання даних навколишнього середовища завжди була важливою задачею для людства. Протягом довгих років люди вчились визначати

температуру, тиск, вологість та інші метрики, за допомогою яких можна прогнозувати погоду, або ж стихійні лиха. Завдяки таким прогнозам можна навіть врятувати життя багатьом людям.

В магістерській дисертації для демонстрації роботи програмної бібліотеки обміну даними між процесами використовується модуль датчика температури та вологості підвищеної точності – DHT22. Це маленький модуль, що містить в собі два сенсори в одному корпусі, результати виміру яких передаються на цифровий блок з аналогово-цифровим перетворювачем (для датчика відносної вологості) і на виході з датчика виходить цифровий сигнал (контакт DATA). Для роботи датчика на контакт VCC подається напруга 3.3-6В. DHT22 має дуже низьке енергоспоживання [16]. Датчик відкалібрований на заводі. Технічні характеристики датчика зображено на таблиці 5.1.

Таблиця 5.1 – Технічні характеристики датчика температури і вологості DHT22

Модель	AM2303
Джерело живлення	3.3-6V DC
Вихідний сигнал	Цифровий
Чутливий елемент	Вимірювання RH – полімерний конденсатор. Вимірювання температури – на базі чіпа DS18B20
Вимірювання вологості	0-100% с похибкою ±2%
Збільшення похибки	±0.5%/рік
Вологість гістерезису	±0.3%
Вимірювання температури	-40...+125°C, похибка ±0.5°C
Взаємозаміна	Повністю взаємозамінювані
Розміри	25.1 x 15.1 x 7.7 мм
Вага	2.2 гр.

“Чутливим елементом датчика вологості ємнісного типу відносної вологості (RH) є полімерний конденсатор. За допомогою даного датчика вимірювання вологості можна проводити у всьому діапазоні (від 0 до 100%), причому похибка вимірювання при використанні нового датчика складає не більше 2%. Як і у всіх датчиків вологості ємнісного типу, з кожним роком конденсатор трохи втрачає свої властивості, що призводить до збільшення похибки. У цьому датчику похибка змінюється на ± 0.5% в рік” [16].

“Датчик температури цифровий, побудований на основі чіпа DS18B20. Він дозволяє вимірювати температуру в діапазоні від -40 до + 125 °C з похибкою ± 0.5 °C” [16].

“В датчика температури і вологості DHT22 є вбудована пам'ять, в якій можуть накопичуватися результати вимірювань і потім, за запитом, передаватися на контролер” [16].

Датчик температури і вологості DHT22, а також відповідність його контактів зображено на рис. 5.4.

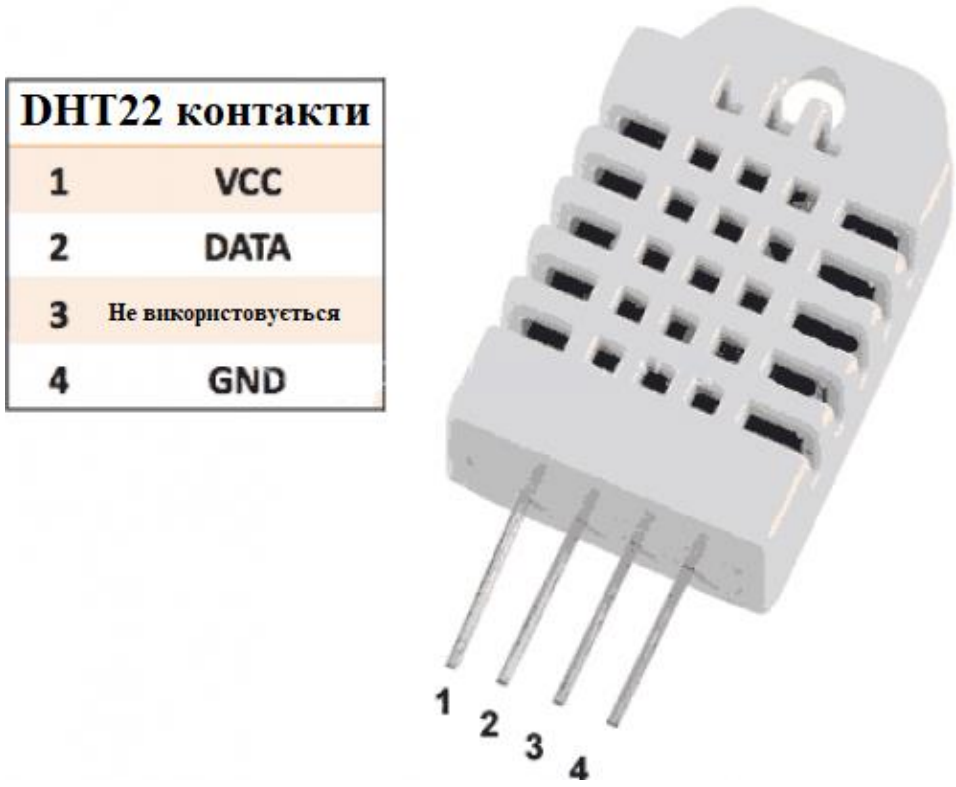

Рисунок 5.4 – Датчик температури та вологості DHT22.

5.3. Схема підключення сенсору до мікроконтролера

Схема апаратної реалізації прикладу використання програмної бібліотеки обміну даними між процесами рухомим об’єктом включає в себе: одноплатний комп’ютер RPi3 та датчик температури та вологості DHT22. Схему зображено на рис. 5.5.

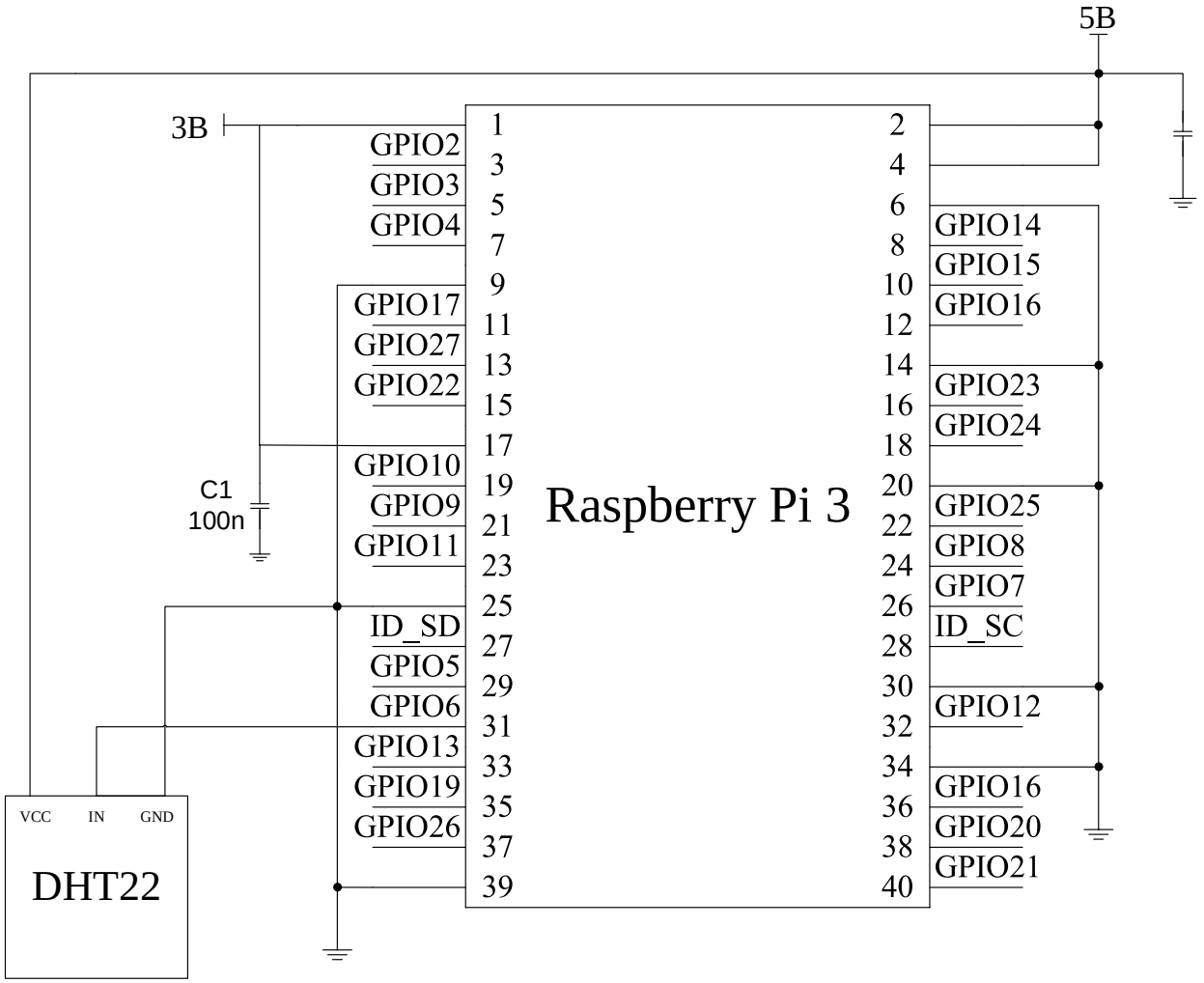

Рисунок 5.5 – Схема апаратної реалізації прикладу використання програмної бібліотеки.

До виходів GPIO Raspberry Pi 3 (рис. 5.5) датчик температури та вологості під’єднується так:

- GPIO6 (31 контакт) – вхідний контакт для отримання даних;
- VCC 5V (2 контакт) – живлення датчику вологості та температури 5В;
- GND (25 контакт) – контакт «земля» (рівень логічного «0»);

Слід зауважити, що для отримання даних з датчику, або ж для під'єднання додаткових сенсорів, можна використовувати будь-який інший вільний контакт GPIO, проте після цього потрібно буде модифікувати програмне забезпечення сторони сервера і додати обробку такого інформаційного входу\виходу.

Так як контакти GPIO напряму під'єднуються до процесору RPi3, потрібно бути уважним і не подати вхіний сигнал з високою силою струму, а також бути обачним і не створити короткого замикання.

5.4. Отримування даних та передача від сервера до клієнта

Для зчитування даних з сенсору DHT22 сервер (RPi3) повинен керувати GPIO, в залежності від того, який сигнал отримано через сокет. Власне читання даних реалізоване в одному модулі, який за допомогою бібліотеки Pi4J передає та отримує сигнали GPIO.

Основним класом Pi4J є GpioController, який отримується в єдиному екземплярі, оскільки він являється синглтоном. За допомогою методу provisionDigitalOutputPin() цього класу отримується клас GpioPinDigitalOutput і відразу встановлює пін в стан OUT. Такий стан означає те, що за допомогою піна не можна зчитувати дані, проте можна виставляти на ньому високий або низький рівень струму. За допомогою такого стану відбувається управління світлодіодом на Raspberry Pi 3 [17]. Екземпляр класу GpioPinDigitalOutput має такі методи для керування станами контакту GPIO:

- high() – встановлює сигнал логічної одиниці на виході;
- low() – встановлює сигнал логічного нуля на виході;
- pulse(long millis, boolean blockingCall) – встановлює сигнал логічної одиниці на час, що вказаний в параметрі millis, після чого встановлюється сигнал логічного нуля;
- toggle() – інвертує логічний стан виходу;

– setShutdownOptions(boolean flag, RaspiPin state) – встановлює логічний стан виходу, вказаний в параметрі state, який буде встановлений після вимкнення GPIO контролера (після завершення програми) [17].

Керування світлодіодом відбувається на виході GPIO 05 (29 контакт). Для цього клієнтський додаток відправляє на сервер булеве значення в якому визначається який рівень повинен бути встановлений на 29 контакті: логічного «0» чи логічної «1».

За допомогою методу provisionDigitalInputPin() класу GpioController повертається екземпляр класу GpioPinDigitalInput і відразу встановлює пін в стан IN. Такий стан означає, що пін використовуватиметься лише для зчитування даних з зовнішніх датчиків, таких як DHT22 [17].

Для отримання коректних даних з модуля датчика температури та вологості DHT22 були створені такі класи DHTxxBase.java та DHT22.java. Вони являються своєрідними адаптерами між нативним кодом мовою C та кодом з вищим рівнем абстракції мовою Java. В разі виникнення помилки при зчитуванні даних з датчика, відбудеться повторне зчитування через 2 секунди. Таких повторних зчитувань може бути 3 рази, після чого виникне помилка DHT22ReadingException.

Програмна бібліотека обміну даними між процесами була створена для розробки складних, розширюваних програмних додатків з широкою функціональністю для того, щоб можна було реалізувати асинхронний обмін даними між процесами на вищому рівні абстракції. Саме тому дуже важливо підібрати правильну архітектуру для такого додатку. В програмно-апаратній моделі продемонстровано використання програмної бібліотеки обміну даними між процесами в межах «чистої архітектури» Роберта Мартіна.

Стаття про «чисту архітектуру» написана Робертом Мартіном ще в 2012 році, проте вона актуальна до цих пір [18]. Дуже багато комерційних проектів, які мають бути надійними та мати мало помилок побудовані саме за допомогою цієї архітектури. Схематично архітектуру зображено на рис. 5.6.

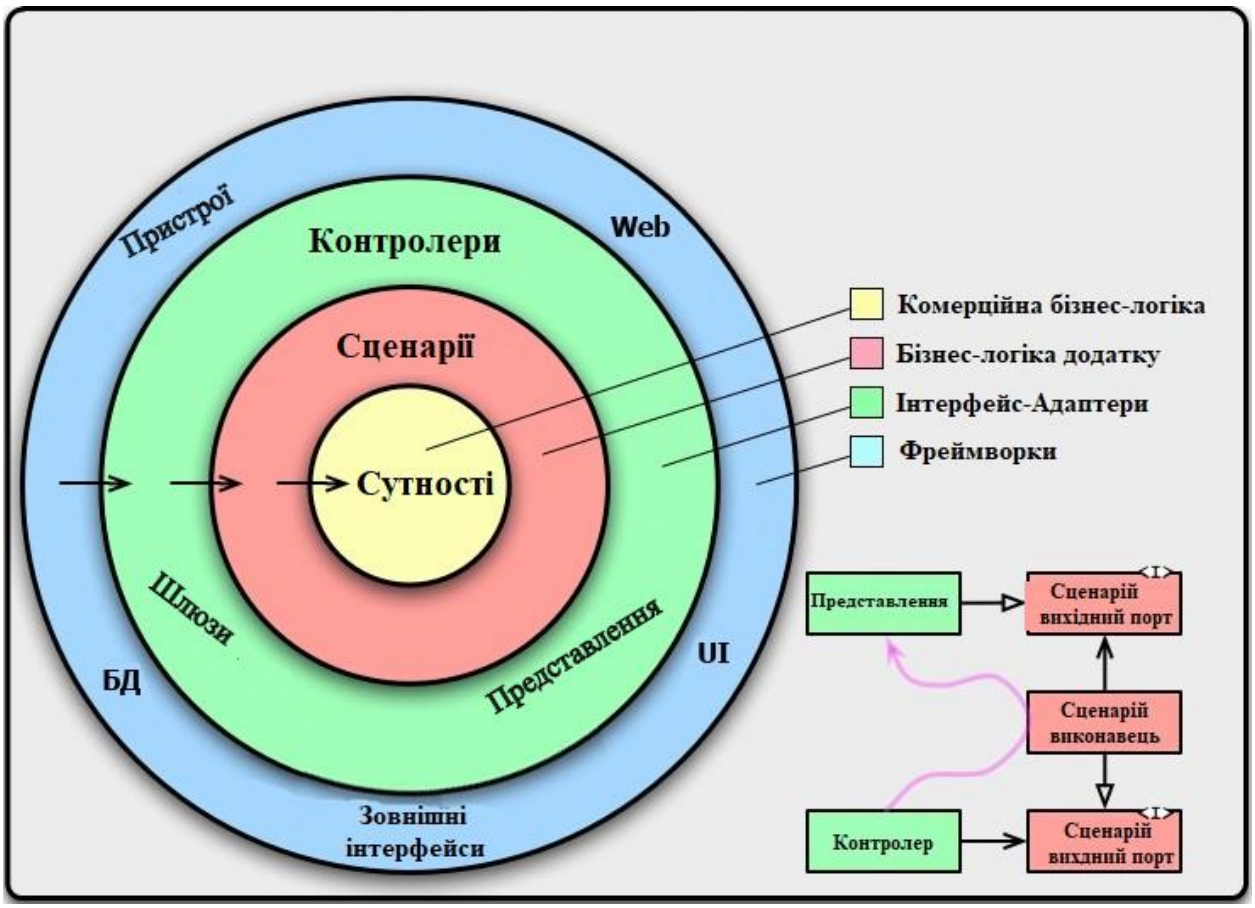

Рисунок 5.6 – Схематичне зображення «чистої архітектури» Роберта Мартіна

«Чиста архітектура» - це така архітектура програмних додатків, яка відповідає таким вимогам:

– незалежність від фреймворків. Не залежить від існування якої-небудь бібліотеки чи фреймворку. В такому випадку це надає можливість використовувати фреймворк як інструмент, при цьому не додаючи свою систему в рамки його обмежень [18];

– тестувальність. Бізнес логіка має мати змогу бути протестованою незалежно від графічного інтерфейсу, бази даних, веб-серверу чи іншого елементу системи [18];

– незалежність від графічного інтерфейсу. Користувацький інтерфейс можна з легкістю змінити, що зазвичай і відбувається після спілкування із замовниками. Наприклад, веб-інтерфейс може бути замінений на мобільну версію програмного додатку. В такому випадку немає сенсу змінювати разом з інтерфейсом і бізнес-логіку [18];

– незалежність від баз даних. Можна змінити базу даних з SQLite на MongoDB, Oragle, MySQL, тощо. Бізнес-логіка не повинні бути зв’язані з базою даних [18];

– незалежність від будь-якого зовнішнього сервісу. Бізнес-логіка нічого не знає про зовнішній світ [18].

Ідея, завдяки якій ця архітектура працює так, як треба, називається *правилом залежностей*. Це правило свідчить про те, що залежності можуть бути спрямовані лише із зовнішніх кругів до внутрішніх. Нічого з внутрішніх шарів не повинно знати про те, що відбувається в зовнішніх шарах. Це стосується функцій, методів, класів, змінних, тощо. В такому випадку ми досягаємо того, що бізнес-логіка не залежить від бази даних, графічного інтерфейсу, зовнішніх сервісів та фреймворків. Структури даних, що створюються та використовуються в шарах не повинні використовуватись в внутрішніх шарах. Як правило такі структури даних дублюються, проте мають деякі відмінності [18].

Сутності визначаються комерційною бізнес-логікою. Сутність – це не лише простий об’єкт, що містить поля та структури даних. Він також може містити методи, що опрацьовують ці поля згідно визначеної бізнес-логіки. Сутності – це такі об’єкти, що малоймовірно зміняться через зовнішні обставини, проте можуть змінитись у випадку зміни бізнес-процесів, а отже зовнішні зміни ніяк не повинні впливати на шар сутностей. Прикладами сутностей є такі об’єкти: комп’ютер, смартфон, автомобіль, людина, тощо.

На шарі сценаріїв відбувається реалізація специфіки бізнес-процесів. Такі сценарії реалізують потік даних від сутностей та до них. Зміни в зовнішніх шарах також не повинні змінювати поведінку сценаріїв, оскільки сценарії тісно співпрацюють з сутностями. Проте, зміни в роботі програмного додатку можуть вплинути на зміни сценаріїв, оскільки вони є немов адаптером між строго визначеними сутностями та реалізацією бізнес-процесів [18].

Програмне забезпечення на шарі інтерфейс-адаптерів являє собою набір адаптерів, що називаються мапери (mapper). Такі механізми перетворюють

дані, що містять сутності в об’єкт, що найбільше підходить, наприклад, для відтворення в базі даних, та навпаки. Код, що знаходиться всередині цього кола не повинен нічого знати про конкретну реалізацію сутностей чи бази даних, проте правильно трансформувати дані згідно визначених інтерфейсів. Якщо база даних – SQL, тоді інтерфейс-адаптери не повинні містити SQL-інструкцій.

Зовнішній шар складається з фреймворків, баз даних, графічного інтерфейсу, тощо. Це такі складові, що містять деталі, неважливі для бізнес-логіки. Зазвичай, багато коду на даному шарі не пишеться, окрім тієї частини, яка спілкується з внутрішніми шарами [18].

Хоча на діаграмі зображено 4 кола, зовсім не обов’язково строго слідувати даним правилам. В залежності від бізнес-правил встановлюється будь-яка кількість шарів, яка найбільше підійде для визначених умов. Проте рівень абстракції завжди має бути меншим на колах ближчих до середини. Також, головним правилом є *правило залежностей,* воно застосовується завжди. Зовнішнє коло – деталі, внутрішнє – найбільш загальна бізнес-логіка.

Правило перетину кордонів між шарами зображено на рис 5.6. в правому нижньому кутку. Видно, що стрілки вказують лише всередину. Потік керування починається з контролера, рухається через сценарій та закінчується в представленні. Зазвичай правило залежностей вирішується за допомогою принципу інверсії керування. Якщо ж необхідно виконати код зовнішніх шарів через внутрішні, в такому випадку необхідно створювати інтерфейси, які будуть реалізовувати класи зовнішніх шарів. Така ж логіка використовується для перетину кордонів між всіма іншими колами діаграми [18].

Кордон перетинають звичайні структури даних. Для цього можна використовувати звичайні поля в аргументах функцій, базу даних. Проте найкращим варіантом є створення так званих Data Transfer Object (DTO) – один із шаблонів проектування для передачі даних між архітектурними шарами. DTO – об’єкт, що створюється на зовнішніх шарах і містить дані у вигляді, що найкраще підходить для визначеного зовнішнього шару

(наприклад, для бази даних). Важливо щоб дані, що передаються між шарами, були ізольованими при передачі через кордони. Не потрібно, щоб структури даних, що передаються через кордони, містили дані, що порушують правило залежностей. Передача даних через кордон завжди повинна бути у форматі зручному для внутрішнього кола [18].

«Чиста архітектура» з самого її початку була розроблена в межах веб-розробки, а саме для створення надійної архітектури високонавантажених серверів. Оскільки в даній магістерській дисертації така архітектура використовувалась для розробки клієнтського додатку на платформі Android, тоді її варто адаптувати. На рис. 5.7. зображено розтин оригінальної діаграми зображеної на рис 5.6.

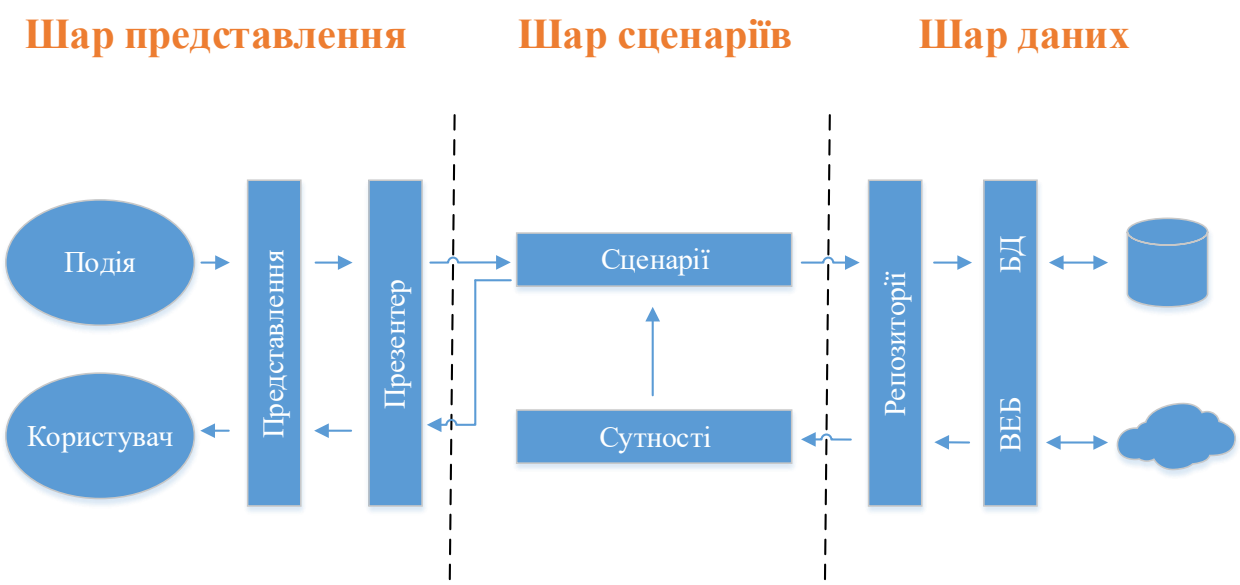

Рисунок 5.7 – Схематичне зображення діаграми «чистої архітектури» в межах платформи Android

Як видно з рис. 5.7, так само як і в оригінальній моделі тут існує 3 шари: шар даних, шар сценаріїв та шар представлення.

В межах платформи Android шар представлення використовує будь-який паттерн відображення (Model-View-Presenter, Model-View-ViewModel, Model-View-Intent, Model-View-Controler, тощо), проте в зображеному прикладі продемонстрований паттерн Model-View-Presenter (MVP). Model – сутності, що стосуються лише шару відображення, їх доцільно використовувати в

програмних додатках з дуже широкою функціональністю, проте вони не використовувались в межах прикладу використання програмної бібліотеки обміну даними між процесами, оскільки є недоцільними, так як повністю дублюють сутності шару сценаріїв. View – це класи, що відповідають за відображення даних на екрані (Activity, Fragments). Presenter – клас, що трансформує дані від шару сценаріїв, та віддає їх до класу View вже в готовому вигляді. Через презентер події передаються через потік до шару сценаріїв.

Шар сценаріїв представляє собою сутності бізнес-логіки. За допомогою класів-сценаріїв відбувається коректна обробка подій та передача даних від них до шару даних, а саме до репозиторію. Репозиторій, в свою чергу повертатиме готові дані через сутності до сценаріїв назад.

Шар даних містить класи репозиторіїв, що визначають яке джерело даних використовувати. Джерелом даних може бути: база даних, віддалений сервер, файл, сокет, тощо. Для кожного такого джерела даних необхідно створювати окремий клас-адаптер для доступу до даних. Варто зауважити, що одні і ті ж дані можуть кешуватись в локальній базі даних, проте оновлюватись через сокет. Саме тому сценарій для обробки одних і тих же даних однаковий. В магістерській дисертації використовувалось лише одне джерело даних – сокет.

Отже, цілком видно що програмна бібліотека чудово вписується в межі «чистої архітектури» Роберта Мартіна, а тому за допомогою неї можна будувати надійний обмін даними між процесами в складних програмних додатках з широкою функціональністю на високому рівні абстракції.

В прикладі використання програмної бібліотеки обміну даними між процесами для забезпечення правила залежностей використовувався контейнер впровадження залежностей Dagger 2. Це один з фреймворків з відкритим сирцевим кодом для впровадження залежностей, який генерує велику кількість шаблонного коду замість розробника. Dagger 2 використовує кодогенерацію, замість рефлексії, а це означає, що він є менш гнучким ніж інші бібліотеки. Проте, простота і надійність згенерованого коду знаходяться на тому ж рівні, що й в написаного власноруч.

Dagger 2 впроваджує залежність сокета, що надсилає та отримує дані від Raspberry Pi 3, а також десеріалізатора та логера для цього. Ініціалізацію зображено в наступному фрагменті програмного коду:

```
@Provides
@ApplicationScope
fun provideSocket(gsonProvided: Gson, loggingInterceptor:
SocketLoggingInterceptor): RxSocket {
    return createRxSocket {
        hostIp = "http://176.36.81.205"
        port = 9092
        namespace = "raspberry"
        socketLoggingInterceptor = loggingInterceptor
        gson = gsonProvided
        options {
            forceNew = false
            reconnection = true
            reconnectionDelay = 1000
            reconnectionDelayMax = 10000
            randomizationFactor = 0.3
        }
    }
}
```

Для логування подій використовується логер Timber, за допомогою нього в контейнері впровадження залежностей ініціалізується SocketLoggingInterceptor:

```
@Provides
@ApplicationScope
fun provideSocketLoggingInterceptor():
SocketLoggingInterceptor {
    return object : SocketLoggingInterceptor {
        override fun logInfo(message: String) {
            Timber.i(message)
        }

        override fun logError(message: String) {
            Timber.e(message)
        }
    }
}
```

В додатку 2 зображені такі джерела даних, що використовують програмну бібліотеку обміну даними між процесами, розроблену в даній магістерській дисертації:

– TemperatureHumiditySocketDataSource – отримує джерело даних Observable, що видає елементи типу TemperatureHumidityDTO (об’єкт, що містить температуру та вологість, отриману від RPi3);

– SocketDataSource – отримує джерела даних, пов’язані з системними подіями (а саме RxSocketEvent). Також містить методи для створення та розриву зв’язку сокета;

– LightsSocketDataSource містить метод для відправки управляючого сигналу світлодіода до Raspberry Pi 3.

Для кожного з цих джерел даних був створений власний репозиторій, оскільки температура\вологість та вмикання\вимикання світла є різними поняттями за бізнес-логікою. В кожному класі репозиторії відбувається трансформація об’єктів з DTO до об’єкта сутності. Тоді клас сценаріїв отримає коректну реалізацію об’єкту даних, при цьому цілісність архітектури не постраждає.

В шарі сценаріїв було створено 3 класи: SocketGenericEventUseCase, TemperatureHumidityUseCase, LightsUseCase. В прикладі, що описаний в магістерській дисертації немає додаткової бізнес-логіки, яка б могла бути оброблена в класах сценаріях, тому вони є тривіальними і просто викликають методи репозиторіїв.

В класі MainActivityPresenter визначається яке повідомлення буде відображено для користувача в залежності від події, яка викликалась сокетом. Після чого реєструється подія отримання даних температури та вологості та створюється з’єднання сокету. При отриманні даних температури та вологості вони відображаються користувачеві на екрані та зберігаються локально. Це зображено в наступному фрагменті програмного коду:

```
override fun initSockets() {
    temperatureHumidityInteractorFacade.onSocketEventListener {
        when (it) {
            CONNECTED -> view()?.showSocketConnected()
            CONNECTING -> view()?.showSocketConnecting()
            DISCONNECTED -> view()?.showSocketDisconnected()
            RECONNECTING -> view()?.showSocketReconnecting()
            ERROR, RECONNECT_ERROR, CONNECT_ERROR ->
```

```
view()?.showSocketError()
            else -> {
            }
        }
    }

    temperatureHumidityInteractorFacade
            .onTemperatureHumidityListener {
                model = it
                view()?.showTemperatureHumidity(it)
            }

    temperatureHumidityInteractorFacade.connect()
}
```

Для відправки керуючих сигналів до сервера в класі MainActivityPresenter були створені методи enableLightsObservable() – для вмикання світлодіода та disableLightsObservable() – для вимикання світлодіода:

```
    override fun enableLightObservable(clickObservable:
    Observable<Any>) {
        addDisposable(clickObservable
                .subscribe {
                    lightsUseCase.execute(true)
                })
    }

    override fun disableLightObservable(clickObservable:
    Observable<Any>) {
        addDisposable(clickObservable
                .subscribe {
                    lightsUseCase.execute(false)
                })
    }
```

Після завершення роботи виконається метод destroy() для закриття з'єднання сокета та вивільнення зайнятої пам'яті:

```
    override fun destroy() {
        temperatureHumidityInteractorFacade.disconnect()
    }
```

На рис. 5.8 зображено знімок екрану готового клієнтського додатку, розробленого в даній магістерській дисертації як приклад використання програмної бібліотеки обміну даними між процесами.

Зліва під піктограмою термометра відображається температура в градусах Цельсію, справа під піктограмою краплі відображається відносна вологість у

відсотках. Над кнопкою «увімкнути світло» відображається текст, який змінюється в залежності від поточного стану сокету. За допомогою кнопки «увімкнути світло» на контакт GPIO 05 подається високий рівень і світлодіод загорається. За допомогою кнопки «вимкнути світло» на контакт GPIO 05 подається низький рівень і світлодіод вимикається.

Рисунок 5.8 – Знімок екрану клієнтського додатку прикладу використання програмної бібліотеки обміну даними між процесами

ВИСНОВКИ

Отже, в межах магістерської дисертації було розроблено програмну бібліотеку обміну даними між процесами, що побудована за допомогою парадигми реактивного програмування мовою Kotlin. Програмна бібліотека може бути використана для реалізації обміну даними в таких програмних додатках, як:

- додатках, що працюють в режимі реального часу;
- чат-додатках;
- IoT-додатках;
- багатокористувацьких іграх.

Запропонована програмна реалізація є альтернативою вже існуючих програмних бібліотек обміну даними між процесами. Проте, її відмінностями є те, що вона відповідає таким вимогам:

- асинхронний обмін даними між процесами;
- простота розробки обміну даними між процесами;
- висока швидкість передачі даних;
- високий рівень абстракції реалізації передачі даних;
- легкість сприйняття програмного коду;
- забезпечення широкої функціональності;
- можливість простого масштабування програмних додатків;
- чутливість кінцевих програмних додатків;
- еластичність кінцевих програмних додатків;
- стійкість кінцевих програмних додатків.

Асинхронний обмін даними забезпечується тим, що для цього використовується технологія, що володіє цією властивістю, а саме сокет. За допомогою нього забезпечується висока швидкість передачі даних при паралельних запитах.

Легкість сприйняття програмного коду та простота розробки кінцевих додатків забезпечується предметно-орієнтованими структурами Kotlin DSL, які були розроблені для ініціалізації екземпляру сокету.

Можливість легкого масштабування програмних додатків забезпечується тим, що програмна бібліотека підтримує розподілення за простором імен, а також реєстрацію подій, визначених розробником. За допомогою таких подій відбувається передача, отримання та класифікація даних.

Коли в дію вступає парадигма реактивного програмування, кінцевий програмний додаток насичується такими властивостями, як чутливість, еластичність та стійкість. За допомогою цих властивостей, програмна бібліотека отримує підтримку легкого масштабування, що є неймовірно важливими при розробці надійних застосунків.

Програмну бібліотеку було розроблено згідно всіх сучасних правил побудови додатків, зокрема були використані такі паттерни проектування, як Observer та Builder. Перечислені паттерни були використані доцільно для покращення читабельності коду і забезпечення принципів GRASP та SOLID. Саме тому програмна бібліотека, розроблена в даній магістерській дисертації вважається актуальною.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Т. Нуркевич, Б. Кристенсен. Реактивное программирование с использованием RxJava. ДМК Пресс, 2017. – С. 6-74.

2. Рейтинг языков программирования 2018: Go и TypeScript вошли в высшую лигу, Kotlin стоит воспринимать серьезно [Електронний ресурс]. – 2018. – Режим доступу: https://dou.ua/lenta/articles/language-rating-jan-2018 – Дата доступу: листопад 2018.

3. Why Kotlin language use is skyrocketing [Електронний ресурс]. – 2018 – Режим доступу: https://appdevelopermagazine.com/why-kotlin-language-use-is-skyrocketing. – Дата доступу: листопад 2017.

4. Stevens, Richard. UNIX Network Programming, Volume 2, Second Edition: Interprocess Communications. Prentice Hall, 1999. ISBN 0-13-081081-9

5. Ю. Берко, О. М. Верес. Організація баз даних: практичний курс : Навч. посіб. для студ. Нац. ун-т "Львів. політехніка". – Л., 2003. – 149 с. – Бібліогр.: 8 назв.

6. О модели взаимодействия клиент-сервер простыми словами [Електронний ресурс]. – 2016. – Режим доступу: http://zametkinapolyah.ru/servera-i-protokoly/o-modeli-vzaimodejstviya-klient-server-prostymi-slovami-arxitektura-klient-server-s-primerami.html. – Дата доступу: травень 2018.

7. Hypertext Transfer Protocol - HTTP/1.1. RFC 2616 [Електронний ресурс]. – Червень 1999. – Режим доступу: https://tools.ietf.org/html/rfc2616. – Дата доступу: листопад 2018.

8. Основа www: протокол HTTP [Електронний ресурс]. Червень 2016. – Режим доступу: http://www.4stud.info/web-programming/protocol-http.html – Дата доступу: листопад 2018.

9. Рой Філдинг. Докторська дисертація на тему Representational State Transfer [Електронний ресурс]. – 2000. – Режим доступу:

https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm – Дата доступу: листопад 2018.

10. Т. Нуркевич, Б. Кристенсен. Реактивное программирование с использованием RxJava. ДМК Пресс, 2017. – С. 6-74.

11. Эрик Фримен, Элизабет Фримен, Кэти Сиерра, Берт Бейтс, Паттерны проектирования. O'Reilly Media, 2013. — 80 с.

12. За что Kotlin так полюбили в Google и кому нужны две тысячи языков программирования [Електронний ресурс]. – 2017. – Режим доступу: http://news.ifmo.ru/ru/science/it/news/6683/. Дата доступу: грудень 2018.

13. Dmitry Jemerov, Svetlana Isakova. Kotlin in action. Manning Publications Co. – 125p.

14. Аналіз ринку смартфонів в Україні [Електронний ресурс]. – 2015. – Режим доступу: http://itc.ua/news/smartfonami-v-ukraine-polzuyutsya-59-polzovateley-v-vozraste-18-30-let-41-v-vozraste-18-50-let/. – Дата доступу: грудень 2018.

15. Raspberry Pi [Електронний ресурс]. – 2018. – Режим доступу: https://www.turkaramamotoru.com/uk/Raspberry-Pi-35991.html. – Дата доступу: грудень 2018.

16. DHT22 datasheet [Електронний ресурс]. – 2016. – Режим доступу: https://cdn-shop.adafruit.com/datasheets/DHT22.pdf. – Дата доступу: грудень 2018.

17. Програмна бібліотека Pi4J. Офіційна документація [Електронний ресурс]. – 2017. – Режим доступу: http://pi4j.com/usage.html. – Дата доступу: грудень 2018

18. Чиста архітектура [Електронний ресурс]. – 2015. – Режим доступу: http://it-ua.info/news/2015/10/27/chista-arhtektura.html - Дата доступу: грудень 2018.